Eötvös Loránd University
Faculty of Informatics
Department of Programming
Languages and Compilers

Ath-Meet Fitness Social Media Application

Supervisor
Ildikó László
Assistant Professor

Author
Na’il Garba
Computer Science BSc

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Student's Data:

Student's Name: Garba Na'il Baba
Student's Neptun code: G86VEI

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *László Ildikó*

Supervisor's Home Institution: *ELTE*

Address of Supervisor's Home Institution: H-1117 Budapest, Pazmany P. setany 1/C, room 2.603

Supervisor's Position and Degree: physicist, PhD, Professor

Thesis Title: Athletes Meet – Social Media App

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

The thesis will consist of a social media app aimed towards athletes. Though people who are not athletes may of course use the app as well. The app will be a twitter style app that allows users to post messages that will appear in other users' feeds. The posts will have the typical features such as likes and comments and sharing between users. Due to the fact that the app is geared towards athletes, the profiles will have features that are important to athletes. Users will have the ability to put their selected gyms on their profile, as well as the sports they usually play, and they can state for how long they've played it, so other users can see their dedication to the sport. There will be an achievements page for users who would like to post their accolades as well. One major feature of the app will be the "find a partner" feature, which will allow users to find other users with similar activities, so that they may potentially workout or train together. This will try and match users with people who are in the same levels of dedication as them as well.
The app will be developed using react native, and it will feature a database that will store users' profile data, as well as the posts and comments. The app will also use an algorithm for the "find a partner" section, that will use profile data to get the best matches.

Budapest, 2021.02.18.

Table of Contents

1. Introduction

The fitness lifestyle is something many developers strive to improve with their own applications. There are only a few platforms that focus on fitness in a social way, which makes fitness a good target for a new social media application. I decided to develop an application that encourages people to share their passion for fitness and provides a meeting ground for people looking for new training partners. The few platforms that focus on fitness in a social way are mainly in the form of online forums, so a social media application would be helpful. A social media style application that encourages people to educate themselves, as well as facilitates the partaking of physical activities that can influence people into leading better lifestyles.

Ath-Meet is intended to be an application designed to help those interested in fitness to socialize with other people who have similar interests and goals. Users of the application will be welcomed with posts on their home screen primarily pertaining to fitness and a healthy lifestyle. These users will be able to interact with each other through posting comments, sharing posts, and connecting with users who are looking for partners in their activities.

Ath-Meet incorporates the standard features that comprise many other social media applications, as well as its key differentiator, the AthleteFinder. These features include creating posts publicly available to all users, a list of posts from other public users, sending private instant messages, following users for quick access to their posts, and having access to the AthleteFinder. The AthleteFinder is a search engine built into the application that lets a user find training partners by taking the user-provided data regarding their main sport, gym, and experience level, and searching for people with matching data. The search engine returns a list of profiles that the user can explore and contact who they connect with the most.

This application will be designed in a way that will be familiar to users of popular social media applications such as Twitter, Instagram, and Facebook. Following the bottom tab navigation style and layout will reduce the make the application feel natural since users already have an understanding of the application's design. Ath-Meet is built using React Native, a Javascript framework built by Facebook, which provides an efficient and simple framework for developing user interfaces for mobile applications [1]. The backend of the application makes use of Amazon Web Services (AWS), a cloud computing platform produced and maintained by Amazon, which provides services such as the Amplify developer toolset, AppSync API, Cognito authentication and user management, DynamoDB database, and S3 storage service [2]. AWS provides services that are highly compatible with React Native, and provides automation and code generation to improve the development process. AWS greatly reduces the work needed to create, maintain, and develop application backends by linking the different services through the Amplify toolset, and providing access to the services through the AWS console that is available through web browsers. The use of a cloud platform greatly reduces the amount of work needed to maintain the database for the application, since servers are already provided that do not have to be set up and maintained.

1.1 Thesis Structure

Chapter 2 contains the User Documentation for the Ath-Meet application. It provides the project description, Sign Up guide, and general information to help the user.

Chapter 3 contains the Developer Documentation which describes the developer software. It details the tools and services used to construct the application, and explains parts of the source code.

2. User Documentation:

2.1 Project Description

The main objective of this project is to create an easily navigable social media application that can also serve as a meeting ground for athletes. Users will be able to create a profile that displays their athletic interests and information about their training, level of experience, and training location. This information is readily available to all users in order to facilitate the meeting process, and provides key information when it comes to the AthleteFinder.

Ath-Meet at its core is still a social media application. A social media application makes use of the interconnections between people to create a social graph that can provide information on groups of people, such as their mutual connections and common interests. These connections can be used to provide a social platform for discussion and information distribution. Ath-Meet constructs this social graph and platform by presenting a list of posts and allowing users to follow each other to communicate. Each post contains text and can feature one image of the user's choice, which can be liked, commented, as well as shared with other users in the private messages to create more discussion. The focus of posts should be to share information about health and fitness for discussions, as well as the achievements of athletes.

The application is targeted at those with an interest in fitness, and meeting people with shared passions for sports and fitness activities. The AthleteFinder is a matchmaker that takes into account a user’s main sport, experience level, and main gym, and finds matching profiles according to these filtered options. It provides a list of profiles that a user can investigate to see who they relate to the most, and allows them an opportunity to connect with them and potentially begin training with them.

Ath-Meet is developed on windows using React Native and Expo. React Native is a framework that offers developers tools for creating mobile applications with JavaScript at its core. It is developer friendly and greatly simplifies the development process of applications by giving templates of common application layouts such as the Bottom Tab Navigator that is used for Ath-Meet. React Native also allows applications to be used across multiple platforms including iOS, Android, and web browsers. Additionally, Expo is used to provide an environment for React Native to connect with community developed packages. These packages are open-source and made to give React Native functions and operations that the standard React Native library does not provide.

For the backend, Amazon Web Services (AWS) provides all the necessary features to store users and data on a cloud platform. AWS provides databases, user management services, and storage for the application, all of which is provided at no cost when there is no large scale data usage.

2.2 Main Elements

The main elements of the application can be summed up as:

- Bottom tab navigation
- Posts
- Comments
- Messages
- Profiles
- Search
- AthleteFinder

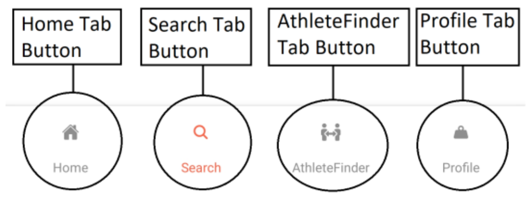
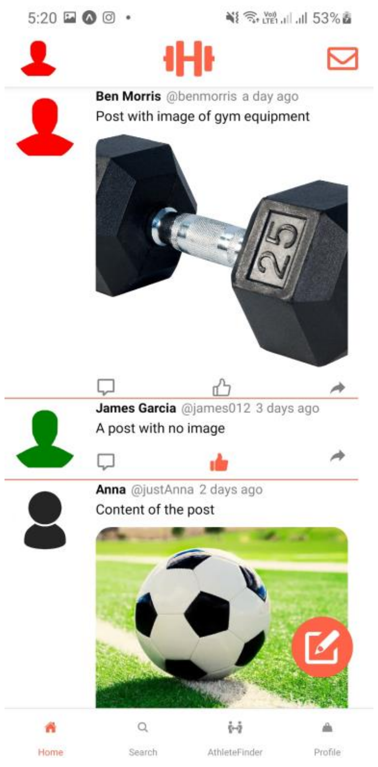


Figure 1: Bottom Tab Navigation Buttons

Ath-Meet uses a bottom tab navigation style so as to facilitate the user experience. Bottom tab navigation provides tabs at the bottom of the screen that navigate to different sections of an application. It is often used in simple applications that don't require many different categories and menus. Many major social media applications such as Reddit, Twitter, Instagram, and Tinder use this feature, so many users will recognize the format immediately. The advantage of using this navigation style is that it simplifies the application into distinct categories, and the main elements can be grouped together in a way that is easy to use. As seen in Figure 1, the bottom tabs are Home, Search, AthleteFinder, and Profile, and the Home tab is the default page after signing in. The Home tab is also the main tab that user's will use, and connects many of the application's main elements together.


Ben Morris @benmorris a day ago
Post with image of gym equipment

James Garcia @james012 3 days ago
A post with no image

Anna @justAnna 2 days ago
Content of the post

Figure 2: Home Tab

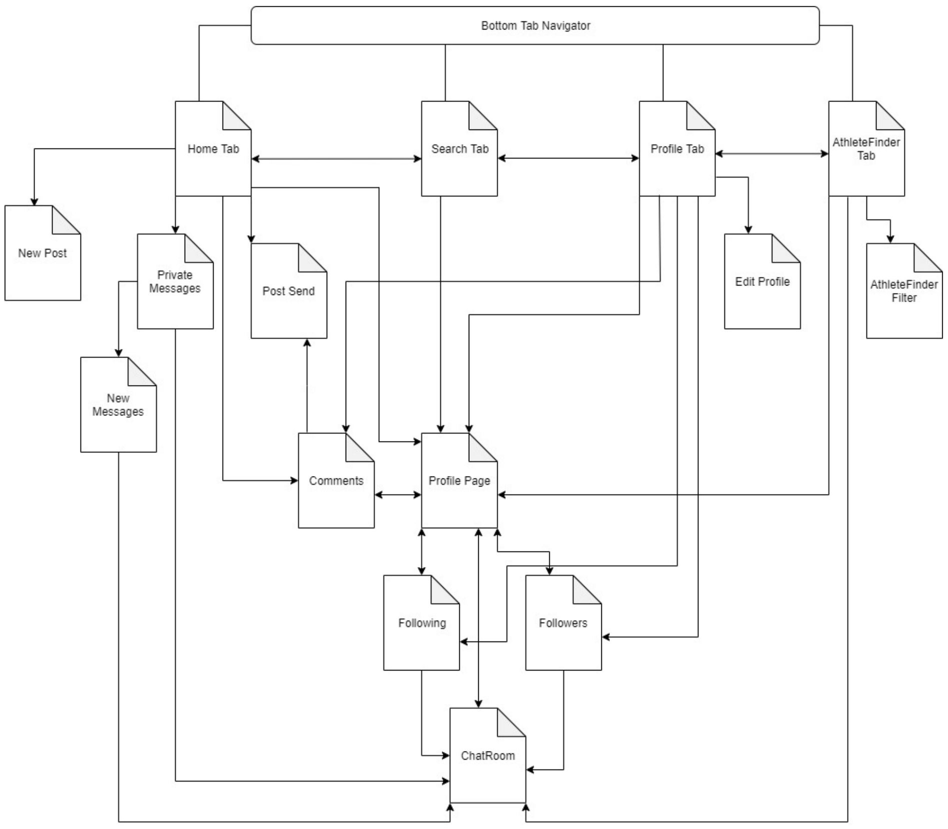


Figure 3: Screen Navigation Diagram

The Home tab houses a list of public posts, and can lead to post creation and the private messages. The list of posts can contain posts from any user, and users can scroll down to continue reading posts that have been pulled from the server, as seen in Figure 2. These posts can lead to other screens where the posts can be shared or commented on. As shown in Figure 3, the Home tab can navigate users to the private messages screen which contains a list of all ongoing conversations that the user is engaged in. This screen can then lead to a chatroom or the creation of new messages.

The Profile tab shows the user their own profile in the way it would be shown to other users. Each profile contains a user's profile name, profile picture, gym, sport, and level. There is also a list of posts they have posted and buttons that will show people they follow and who are people following them. Users can quickly make changes to their profile from this tab by navigating to the edit profile screen where the user may change name, profile picture, gym, sports, and experience level.

The Search tab will offer the user the option to search and view the profile pages of others by typing in their profile name. It shows a list of users under the search bar, displaying their profile information, and refreshes the list of users automatically as the user types in the name.

The AthleteFinder tab will be a unique feature of Ath-Meet which caters to those looking for partners in particular sports and activities. In this tab, users will be shown a list of profiles that match the AthleteFinder filter criteria as set by the user. Each user can choose their own gym and sport, and choose one of four experience levels; beginner, intermediate, advanced, and expert. By default it will show the profiles that identically match the details that the user has saved to their profile, and in the list of results, this information is displayed for each profile. The AthleteFinder is meant to connect people who may have similar training experience, so the level is important here, as those who have been practicing an activity for a longer time are more likely to want to train with people at the same level. Furthermore, the page features a button that navigates to the AthleteFinder filter screen where the user will be able to change the search criteria for the AthleteFinder, and this may be a change in sport, experience level, or gym.

2.3 Usage Information

2.3.1 Getting started

In order to use Ath-Meet, users must have a registered Ath-Meet account. Launching the application will take users that have not signed in to the sign in screen, through which they may choose to create an account if they do not have one or sign in to an existing account. Only after entering their credentials and being authenticated will the user be able to access the application.

2.3.2 Registration and Forgot Password

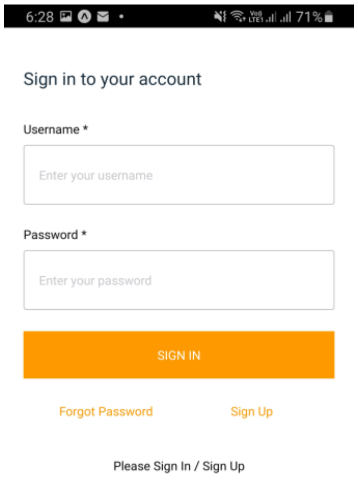


Figure 4: Sign In Screen

Upon launching Ath-Meet, users are greeted with the Sign In screen as seen in Figure 4, and are prompted to enter a username and password to sign in or choose to sign up which will navigate to the Sign Up screen.

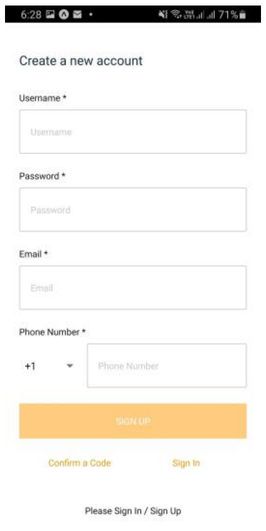


Figure 5: Sign Up Screen

As seen in Figure 5, the Sign Up screen asks for an email, username, password, and phone number in order to create a new account. The user will be sent a code to their email and phone number which will be used to verify them before they are registered. The phone number is only used for verification upon registration, and is not saved in the database. By creating an account, a new user is registered in the Ath-Meet database. Users can sign in across multiple devices simultaneously and will have their data linked and updated together.

For users who have been registered, entering their correct credentials in the Sign In screen will send their credentials for authentication . If the authentication is successful, the user will be sent to the Ath-Meet home screen. If authentication fails, they will receive an error message, and the user may choose to reset their password or sign up for a new account.

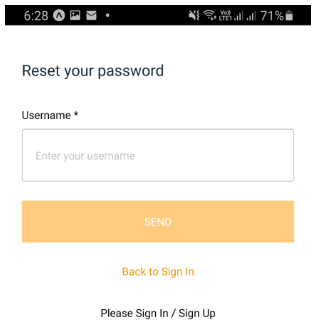


Figure 6: Reset Password Screen

The Reset Password screen gives users with existing accounts that have lost their password a chance to reset it. Pressing on the button will navigate to the password reset screen where users will be asked to enter their username so that an email can be sent to them with a link to reset their password.

2.3.3 Home Tab

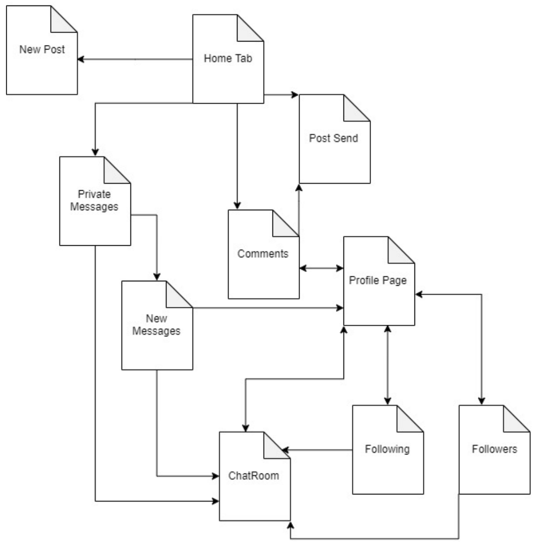


Figure 7: Home Tab Navigation Diagram

The Home tab is the main tab and has the most connections to screens among the tabs. Once a user is signed in, they will be greeted by the Home tab which features a list of posts from all users on the application.

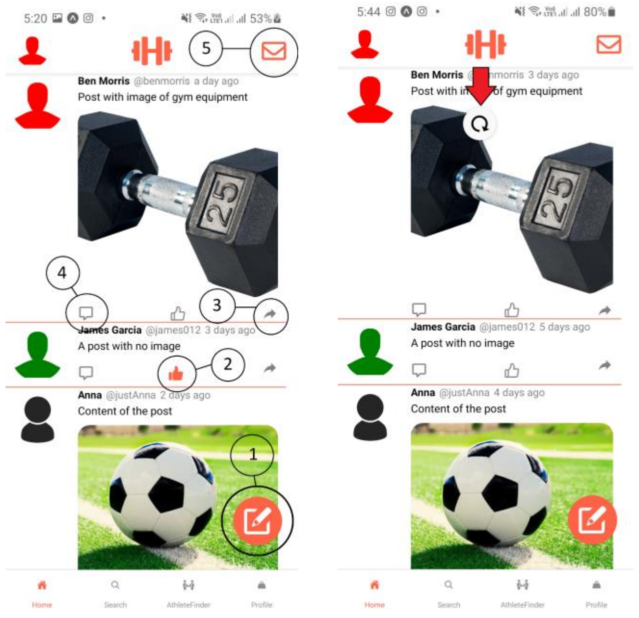

5
4
3
2
1
Ben Morris @benmorris a day ago
Post with image of gym equipment
James Garcia @james012 3 days ago
A post with no image
Anna @justAnna 2 days ago
Content of the post
Home
Search
AthleteFinder
Profile

Figure 8: Home Tab showing a list of posts, and pulling down to refresh Home Tab.

When the user is at the top of the list of posts, they can pull down on it to refresh it, as seen in Figure 8. The Home tab can navigate directly to the new post (button 1), share post (button 3), comments (button 4), and messages (button 5) screens.

2.3.3.1 Posts, Likes, and Shares

The primary form of communication on Ath-Meet is the post. It features text, and an optional image uploaded by the person who posted it which is then displayed publicly on their profile and in other user’s Home tabs. Each post can be commented on and liked by any user that comes across the post. Additionally, users can choose to share posts privately in their chats with others.

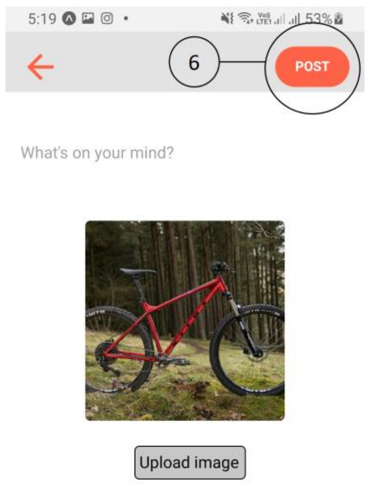


Figure 9: New Post Screen

To make a post, the user must navigate to the Home tab, and press button 1. This will navigate them to the new post screen where they will be able to type in the content of the post, and provide an image to be attached along with it. Once a user is satisfied with the post, they can publish it by pressing button 6 which will send the user back to the Home tab and publish the post, as seen in Figure 9. They can refresh their screen to see the post, or continue scrolling down the list of posts.

Each post has the option to be liked by a user which is done by pressing button 2. A liked post will have a button with a red color. The post can be unliked as well which will revert the button to its original gray color.

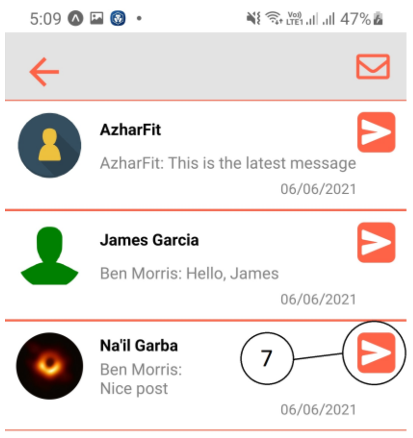

AzharFit
AzharFit: This is the latest message
06/06/2021
James Garcia
Ben Morris: Hello, James
06/06/2021
Na'il Garba
Ben Morris: Nice post
06/06/2021

Figure 10: Share Post Screen

Each post can also be shared by pressing button 3. This button navigates to a screen that shows the user's private messages where they can choose to send a post to a chat by pressing on button 7, as seen in Figure 10.

2.3.3.2 Comments

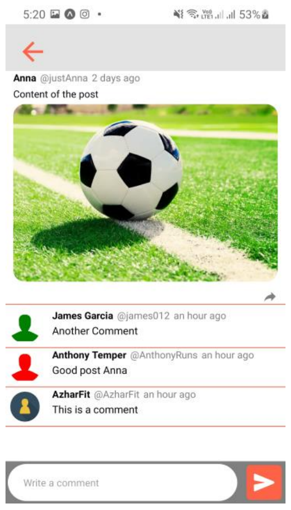
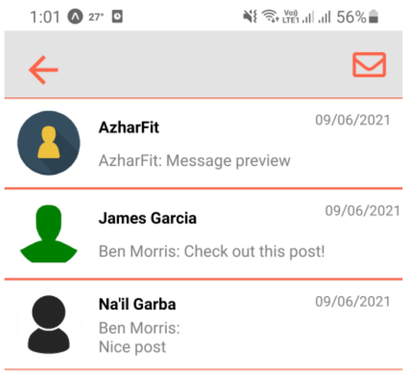

Anna @justAnna 2 days ago
Content of the post
James Garcia @james012 an hour ago
Another Comment
Anthony Temper @AnthonyRuns an hour ago
Good post Anna
AzharFit @AzharFit an hour ago
This is a comment

Figure 11: Comments Screen

Button 4 is featured on each post and navigates to the comments screen for the post when pressed. The comment screen features the post at the top of the screen and also displays a list of comments that have been made under the post, as shown in Figure 11.

Comments do not feature images, and only contain text as their content. Pressing on the profile picture of a comment navigates to that user's profile page.

2.3.3.3 Private Messages and Chatrooms

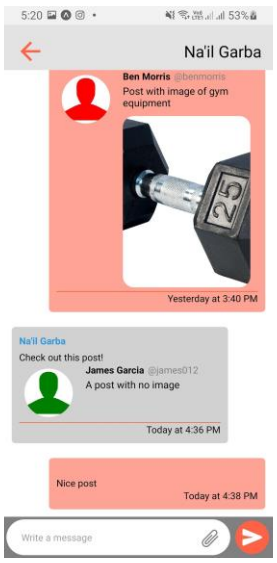

1:01 27° 56%

AzharFit 09/06/2021
AzharFit: Message preview

James Garcia 09/06/2021
Ben Morris: Check out this post!

Na'il Garba 09/06/2021
Ben Morris:
Nice post

Figure 12: Messages Screen


5:20 53%

Na'il Garba

Ben Morris @benmorris
Post with image of gym equipment
Yesterday at 3:40 PM

Na'il Garba
Check out this post!
James Garcia @james012
A post with no image
Today at 4:36 PM

Nice post
Today at 4:38 PM

Write a message

Figure 13: Chatroom Screen with sent posts

Pressing the button 5 will navigate to the private messages screen where a user will see all of his existing chats with others, as shown in Figure 12. Pressing on a chat will navigate the user to the selected chatroom through which two users can send instant messages and posts to each other, as seen in Figure 13.

2.3.4 Profile Pages and Profile Tab

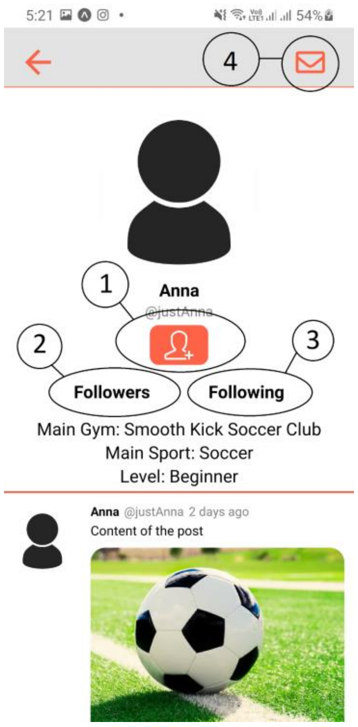


Figure 14: Profile Page

A profile page contains all the details of the user's profile information and their posts, and the Profile tab displays the user’s personal profile page, as seen in Figure 14 and 15. The profile information includes the user’s profile picture, profile name, username, main gym, main sport, and experience level of their main sport. Under the profile information is a list of posts that have been posted by the user.

When a user clicks on someone's profile picture anywhere in the application, they will be navigated to that user's profile page. On profiles that are not being followed by the user, button 1 will show an outline of a person with a plus sign on the right, and pressing this will make the user follow the profile and add the profile to the following list of the user and the user to the profile's followers list. When a user is being followed, the button will show a checkmark instead of the plus sign, and pressing this will undo the previous actions. Users can press button 2 to see a list of users that follow the profile. Similarly, pressing button 3 will show a list of users that are being followed by the profile. Pressing button 4 will navigate the user to a new chatroom with the profile, providing a quick way to privately message users.

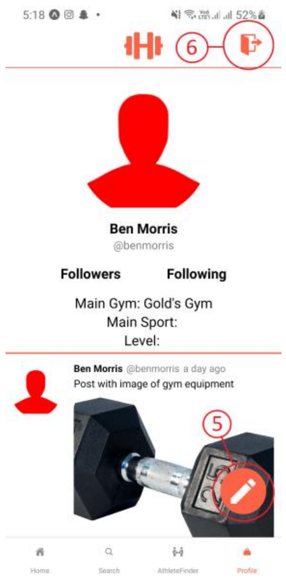


Figure 15: Profile Tab

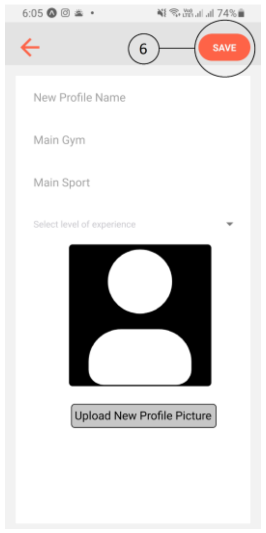
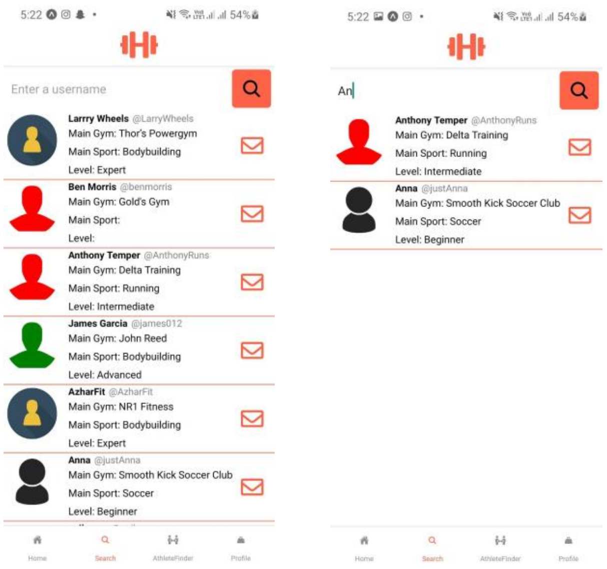


Figure 16: Edit Profile Screen

The Profile tab displays a user's personal private page, and they can choose to edit it by pressing button 5, as shown in Figure 15. This navigates users to the edit profile screen where they can alter their profile information, as seen in Figure 16. Once the user is done editing, they can press button 6 to save the changes. The display picture for new users is set by default to a profile icon outline, and their profile name is set to their username. All other fields are left blank until the user changes them. Pressing button 6 will sign the user out of Ath-Meet, and users must sign in again to regain access to the application.

2.3.5 Search Tab



Larrry Wheels @LarryWheels
Main Gym: Thor's Powergym
Main Sport: Bodybuilding
Level: Expert

Ben Morris @benmorris
Main Gym: Gold's Gym
Main Sport:
Level:

Anthony Temper @AnthonyRuns
Main Gym: Delta Training
Main Sport: Running
Level: Intermediate

James Garcia @james012
Main Gym: John Reed
Main Sport: Bodybuilding
Level: Advanced

AzharFit @AzharFit
Main Gym: NR1 Fitness
Main Sport: Bodybuilding
Level: Expert

Anna @justAnna
Main Gym: Smooth Kick Soccer Club
Main Sport: Soccer
Level: Beginner

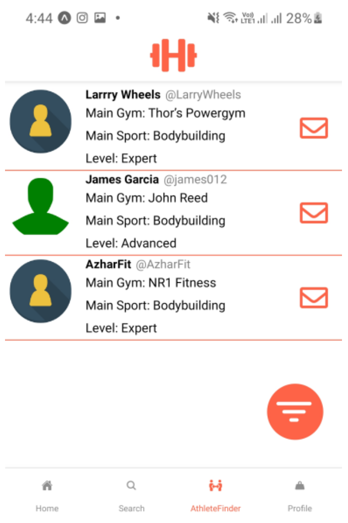
Anthony Temper @AnthonyRuns
Main Gym: Delta Training
Main Sport: Running
Level: Intermediate

Anna @justAnna
Main Gym: Smooth Kick Soccer Club
Main Sport: Soccer
Level: Beginner

Figure 17: Search Screen

The Search tab gives users the ability to search for profiles by their profile name. It contains a text input bar at the top of the page where users can type in names of the profiles, and the search is case sensitive. The search tab automatically updates as a user is typing. The list of profiles displays the profile information of each profile and also contains a message button on the right of each profile which will navigate the user to the appropriate chatroom, as seen in Figure 17.

2.3.6 AthleteFinder Tab


4:44 28%

Larrry Wheels @LarryWheels
Main Gym: Thor's Powergym
Main Sport: Bodybuilding
Level: Expert

James Garcia @james012
Main Gym: John Reed
Main Sport: Bodybuilding
Level: Advanced

AzharFit @AzharFit
Main Gym: NR1 Fitness
Main Sport: Bodybuilding
Level: Expert

Figure 18: AthleteFinder Screen

The AthleteFinder is a feature that distinguishes Ath-Meet from its social media competitors, by focusing on finding people that share interests with the user. In this tab, the user is presented a list of users who have the same main sport the user has saved on their profile. This list shows each profile's main gym, sport and level, and allows the user to visit the profiles to see if they are interested in contacting the person, as seen in Figure 18.

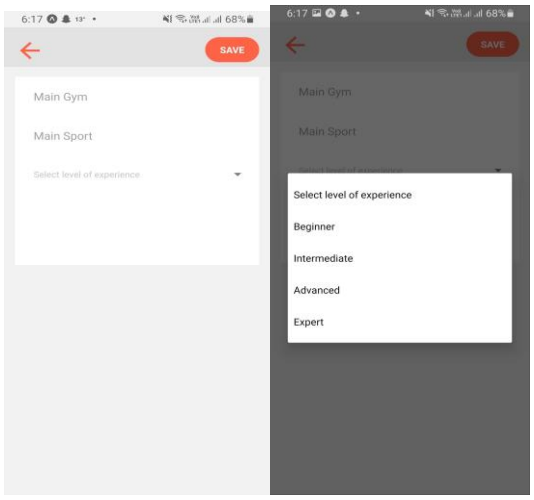


Figure 19: AthleteFinder Filter Screen

The filter button on the bottom right navigates to the AthleteFinder filter screen where a user can change the criteria that the AthleteFinder searches for. As shown in

Figure 19, the main gym, sport, and level can all be changed, and if the user does not want to filter results on a particular criteria, they can leave it blank. The new filter settings can be saved by pressing “save” which will refresh the AthleteFinder tab with new users that match the new search criteria. This tab is meant to bring users together so they can share their passion for fitness and potentially train together.

3. Developer Documentation

3.1 Project Objectives

The main objective of this project is to create a platform for users to interact and communicate with each other having the goal of finding training partners through a mobile application. The challenges in the project include the following:

- Proper integration of the necessary Amazon Web Services including Cognito, DynamoDB database, and S3 storage.
- Providing a reliable and secure user authentication system with AWS Cognito.
- Creating an intuitive user interface through React Native.
- Using Expo libraries and packages to improve React Native’s functionality.
- Allowing users to easily create chats between each other to send instant messages.
- Allowing users to create posts that can contain an image.
- Storing images in AWS S3 storage.
- Storing relevant application data in the AWS DynamoDB database.
- Allowing users to share posts in messages.
- Allowing users to comment and like posts.
- Enabling users to instantly alter their profiles.
- Granting users a means to easily view other user’s profiles.
- Creating a functioning GraphQL schema used by the client to query and upload precise data.
- Constructing a search system for finding user profiles.
- Constructing the AthleteFinder search system to allow users to find potential training partners.

3.2 Front End

The front end of the application is constructed using the React Native JavaScript framework for mobile application development on the Expo platform which provides additional libraries for React Native projects.

3.2.1 React Native

React Native is a JavaScript framework that is used to create mobile applications, and is based on the standard React user interface library. It makes use of JSX, React’s own JavaScript syntax extension that makes it possible to use HTML in combination with JavaScript in React code. This makes the application more dynamic since it allows functions to be used in HTML code by using JavaScript code surrounded by curly braces inside the HTML expressions. Additionally, this allows for the use of CSS for the styling of the user interface. The user interface is built by HTML and CSS, while the program uses JSX for the logic and to structure the application.

React works by structuring an application into components that call and pass down information to child components. Much of the information is held in component states, which can be set to cause many different reactions in components upon these state changes. The states can also be passed or bound to child components which makes the applications even more dynamic and responsive. The division of code into components also makes it easy to change small parts of the user interface without needing to re-render other parts that are unaffected, thus making more efficient use of resources.

Using React also allows for the same source code to be used to develop mobile applications for multiple platforms, including Android, iOS, and the Web. This makes it very versatile for application development and drastically cuts down work time for multiplatform development.

3.2.2 Expo

This project uses Expo in conjunction with React Native to develop a responsive and interactive user interface that uses more features than provided by the standard React Native library. Expo also provides the Bottom Tab Navigation template for React Native that was used in this project. By virtue of using Expo, this project can make use of many open source packages developed by the React Native community [3]. Ath-Meet uses community developed packages such as Picker Select for dropdown menus, Safe Area Context to define proper screen boundaries for different platforms, and Gesture Handler for implementing touchable opacities. Expo uses the Node.js runtime environment, so Node.js must be installed on a computer when developing with Expo. All the Expo packages are installed by calling “expo install,” “npm install,” or “yarn add” followed by the name of the package. To debug run Ath-Meet, the Expo Command Line Interface (CLI) must be installed on the computer, and then the command “expo start” can be called in the terminal to launch the application. To run the application on an Android device, download “ath-meet-android-installation-file.apk” onto the device, open the file and allow the device to install the application.

3.2.3 Implementation of Bottom Tab Navigator

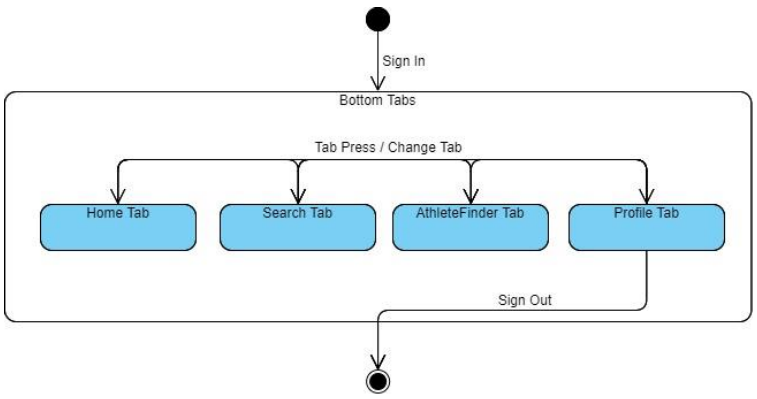
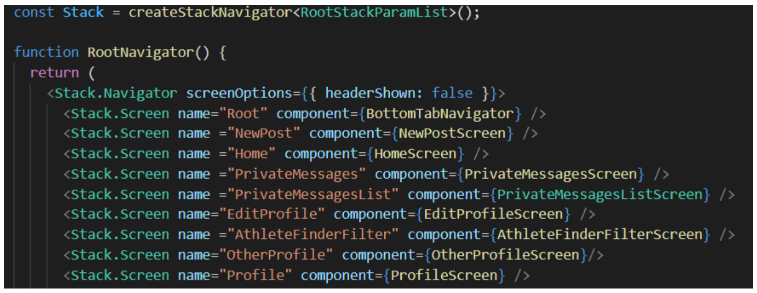


Figure 20: Ath-Meet Bottom Tab Navigator State Diagram

The bottom tab navigation is created from a template provided by Expo, and structures the application by setting screens into different tabs that can switch between each other, as seen in Figure 20. The bottom tab navigation code is stored in the “navigation” folder which contains the index.tsx and BottomTabNavigator.tsx files that need to be configured for bottom tab navigation.

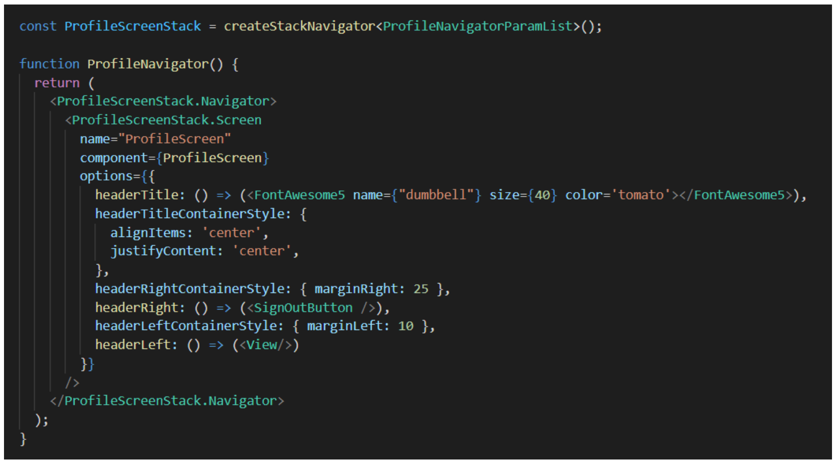


```
const Stack = createStackNavigator<RootStackParamList>();

function RootNavigator() {
  return (
    <Stack.Navigator screenOptions={{ headerShown: false }}>
      <Stack.Screen name="Root" component={BottomTabNavigator} />
      <Stack.Screen name ="NewPost" component={NewPostScreen} />
      <Stack.Screen name ="Home" component={HomeScreen} />
      <Stack.Screen name ="PrivateMessages" component={PrivateMessagesScreen} />
      <Stack.Screen name ="PrivateMessagesList" component={PrivateMessagesListScreen} />
      <Stack.Screen name="EditProfile" component={EditProfileScreen} />
      <Stack.Screen name ="AthleteFinderFilter" component={AthleteFinderFilterScreen} />
      <Stack.Screen name="OtherProfile" component={OtherProfileScreen}/>
      <Stack.Screen name="Profile" component={ProfileScreen} />
```

Figure 21: RootNavigator() in navigation/index.tsx

In RootNavigator() in the index.tsx file, ‘stack navigator screens’ are created using the component for each screen the application must navigate to. The navigator stores screens in a stack to keep track of the navigation, as seen in Figure 21.



```
const ProfileScreenStack = createStackNavigator<ProfileNavigatorParamList>();

function ProfileNavigator() {
  return (
    <ProfileScreenStack.Navigator>
      <ProfileScreenStack.Screen
        name="ProfileScreen"
        component={ProfileScreen}
        options={{
          headerTitle: () => (<FontAwesome5 name={"dumbbell"} size={40} color='tomato'></FontAwesome5>),
          headerTitleContainerStyle: {
            alignItems: 'center',
            justifyContent: 'center',
          },
          headerRightContainerStyle: { marginRight: 25 },
          headerRight: () => (<SignOutButton />),
          headerLeftContainerStyle: { marginLeft: 10 },
          headerLeft: () => (<View/>)
        }}
      />
    </ProfileScreenStack.Navigator>
  );
}
```

Figure 22: ProfileNavigator() in navigation/BottomTabNavigator.tsx

The BottomTabNavigator.tsx file holds the BottomTabNavigator function which loads screens to the specified tabs, and creates special stack navigators for each bottom tab. Each bottom tab’s navigator is constructed in its own React component, and returns the stack screen to be navigated to, as shown in Figure 22.

3.3 Backend

3.3.1 Amazon Web Services and Amplify Command Line Interface

The backend for Ath-Meet relies on Amazon Web Services (AWS) which provides us with a cloud storage and database platform and API service that can be easily integrated with React Native applications. This application uses the GraphQL data query language which works with AWS AppSync API to connect with the DynamoDB database and S3 storage. User authentication services such as the sign in and sign out processes are automatically handled through Amazon Cognito, the user identity authorization and synchronization service, and is managed through Cognito User and Identity Pools on the AWS console. Cognito works by signing users into a user pool and sending the user a token that can be exchanged for AWS credentials through an identity pool [4]. The identity pool is further managed by AWS Identity and Access Management (IAM), which allows for policies and roles to be set to the identity pool users [5].

Amplify Command Line Interface (CLI) is the developer tool that is used for developing Ath-Meet in order to install and manage the various AWS cloud services that are used through command line commands [6]. Amplify includes support for React Native so they work well together, and Amplify's code generation generates code that can be in Typescript and JavaScript. Amplify is used every time a database change occurs and needs to be pushed or pulled between the DynamoDB and the project. It also provides us with the option to generate a GraphQL schema and will generate GraphQL code whenever changes occur in the schema.

Amplify and AWS AppSync work together to manage the backend through the "AWS Management Console" web application. This application provides a user-friendly interface that allows us to examine and modify the schema, functions, mutations, subscriptions, and queries, and manage various API settings for our project. It also provides an interface for Cognito and DynamoDB, giving us full control of the backend. The Identity and Access Management (IAM) system is another service provided by AWS which allows users in the IAM system to be granted abilities for accessing and modifying parts of the backend. Users can have different permissions according to varying policies and roles. Administrator access grants all permissions to a user, giving them the privilege to view, modify, and delete parts of the backend. The IAM users are not connected to Cognito, so users created through Ath-Meet do not have the ability to make changes to the backend. The Cognito users have permissions to create and change items which are stored in the database and storage, and have no ability to delete anything from the database or storage directly in order to maintain security and authenticity of the data.

AWS Amplify makes implementing the entire authentication process very simple in React Native projects, since it auto generates a significant amount of the code for us. First, we install the Amplify package in the source code directory by calling "amplify init" in the console. Once Amplify is installed, we can use it to add all the other AWS services that the project needs by calling the appropriate Amplify command line commands in the terminal. For example, "Amplify add auth" will install all the necessary dependencies from 'aws-amplify' onto the React Native project. This step only adds in the necessary files needed to connect to the Cognito user pool.

Amplify provides us with the ability to add a GraphQL or Rest API to our project. For this project, GraphQL is chosen, since it allows the entire data request process to be handled by the client side, and allows for strict data management. This works well in React Native, since the use of components creates a downwards data flow that can be structured well with predetermined data requirements as set in the GraphQL schema. The schema is a type system that defines the functionality and connections between the data of a GraphQL dataset. [7]. Amplify can generate a sample GraphQL schema for the project which can be used as a starting point for development. The schema for Ath-Meet is based on the blog post sample schema generated by Amplify, with added support for chatrooms, messages, comments, and user followings, and is found in the amplify/backend/api\dev/schema.graphql file.

3.3.2 Authentication with Cognito

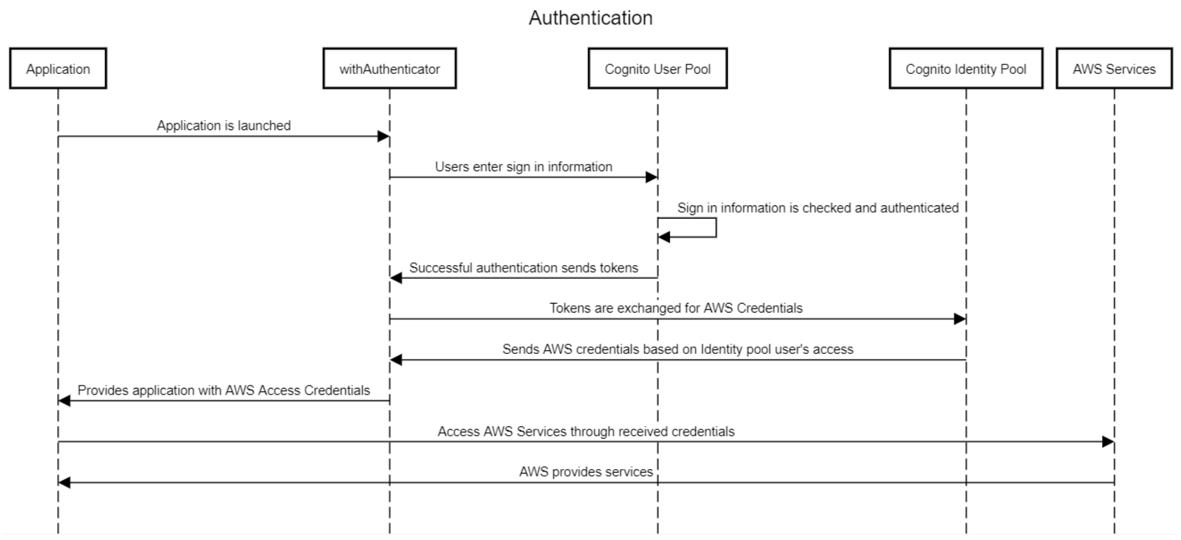

Authentication
Application
withAuthenticator
Cognito User Pool
Cognito Identity Pool
AWS Services
Application is launched
Users enter sign in information
Sign in information is checked and authenticated
Successful authentication sends tokens
Tokens are exchanged for AWS Credentials
Sends AWS credentials based on Identity pool user's access
Provides application with AWS Access Credentials
Access AWS Services through received credentials
AWS provides services

Figure 23: Authentication Sequence Diagram

Authentication is also handled by AWS through Cognito. During sign up processing, the application sends the user credentials to the Cognito user pool for verification and authentication. Once the authentication is complete, the user receives a token which is used to connect to the AWS API cloud gateway, and therefore the application can interact with the AWS services such as S3 and DynamoDB, as shown in Figure 23.

For authentication, the App.tsx file must be slightly modified in order to allow AWS to wrap the application. We do this by configuring Amplify and the API, and then exporting React Native’s default App function by passing it through the AWS withAuthenticator() function, and this will make the entire application enclosed by AWS authentication.

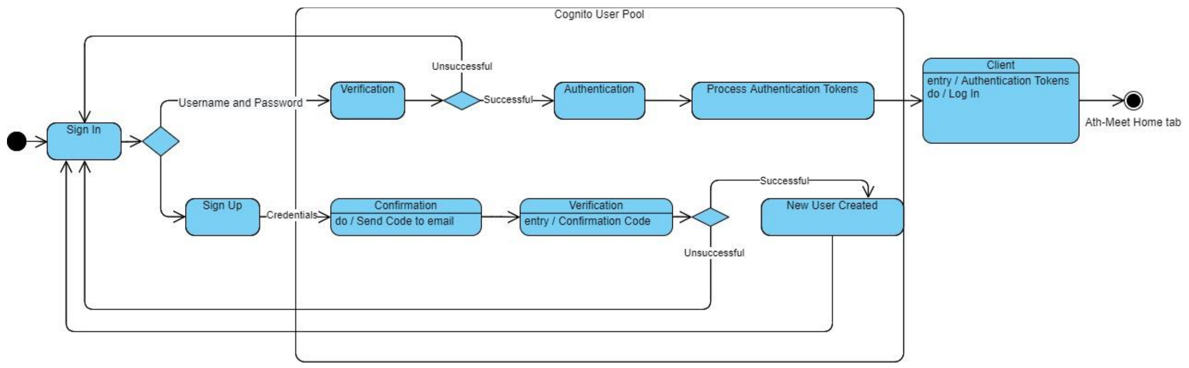
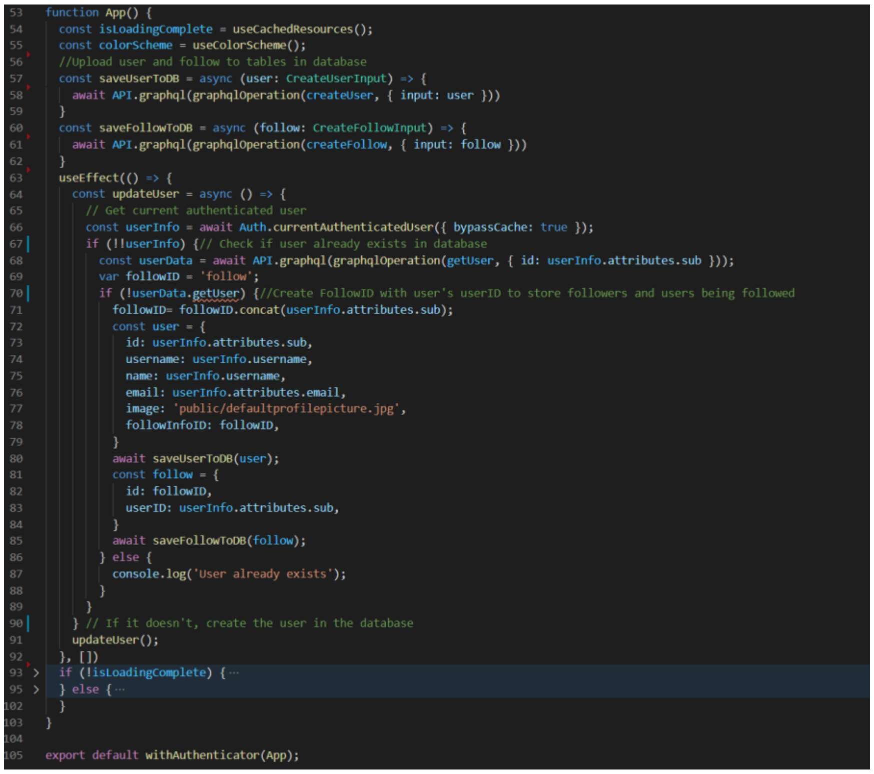


Figure 24: Authentication Action Diagram

For registration, a user must sign up in the application by providing their username, name, phone number, and email. After that, a verification code will be sent through email and the user must enter if before user creation, as shown in Figure 24. In the App.tsx, a new user variable will be created with the Cognito user information, excluding the phone number, and then the application calls the “saveUserToDB” GraphQL mutation to create a new user profile in the database.



```
function App() {
  const isLoadingComplete = useCachedResources();
  const colorScheme = useColorScheme();
  //Upload user and follow to tables in database
  const saveUserToDB = async (user: CreateUserInput) => {
    await API.graphql(graphqlOperation(createUser, { input: user }))
  }
  const saveFollowToDB = async (follow: CreateFollowInput) => {
    await API.graphql(graphqlOperation(createFollow, { input: follow }))
  }
  useEffect(() => {
    const updateUser = async () => {
      // Get current authenticated user
      const userInfo = await Auth.currentAuthenticatedUser({ bypassCache: true });
      if (!!userInfo) {// Check if user already exists in database
        const userData = await API.graphql(graphqlOperation(getUser, { id: userInfo.attributes.sub }));
        var followID = 'follow';
        if (!userData.getUser) {//Create FollowID with user's userID to store followers and users being followed
          followID= followID.concat(userInfo.attributes.sub);
          const user = {
            id: userInfo.attributes.sub,
            username: userInfo.username,
            name: userInfo.username,
            email: userInfo.attributes.email,
            image: 'public/defaultprofilepicture.jpg',
            followInfoID: followID,
          }
          await saveUserToDB(user);
          const follow = {
            id: followID,
            userID: userInfo.attributes.sub,
          }
          await saveFollowToDB(follow);
        } else {
          console.log('User already exists');
        }
      }
    } // If it doesn't, create the user in the database
    updateUser();
  }, [])
  if (!isLoadingComplete) {⋯
  } else {⋯
  }
}

export default withAuthenticator(App);
```

Figure 25: App() function with user authentication and creation

The user's database userID will be its primary key, and it is used to match with the Cognito userID found in the userInfo.attributes.sub so that the primary keys in both databases are identical. The new user variable is initialized with all necessary attributes that have already been registered in Cognito. As result, the user record is then created in the database, as shown in Figure 25.

3.3.3 GraphQL

GraphQL is a data query language for the API used by Ath-Meet. It uses a schema which is a type system that defines the dataset's connections, and allows us to query data [7]. GraphQL queries return an object that takes in the same structure as the query. The structure of GraphQL queries makes development of components and their manipulation of data much simpler. Subscriptions are queries that can update their result over time, and retain a connection to the server until they are closed. We use subscriptions to fetch information that needs to be updated in real time such as messages. Mutations allow us to change data, and they also can return data after the mutation that allows us to check the results of the mutation immediately. The GraphQL operations generated by Amplify are stored in the “src/graphql” folder in the source code.

The GraphQL schema allows us to create queries that fetch detailed information with optional filters, sorts, or limits. This also brings a downside to GraphQL, as the queries can become complex due to the nesting of objects in the queries. Queries can fetch unnecessary or convoluted data when they are not properly defined, so pushing down necessary information to React Native components can become a challenge. Ath-Meet’s customized queries are located in the “customgraphql” folder. These queries optimize the fetching of user information, posts, chatrooms, and follower information, by querying only the necessary information.

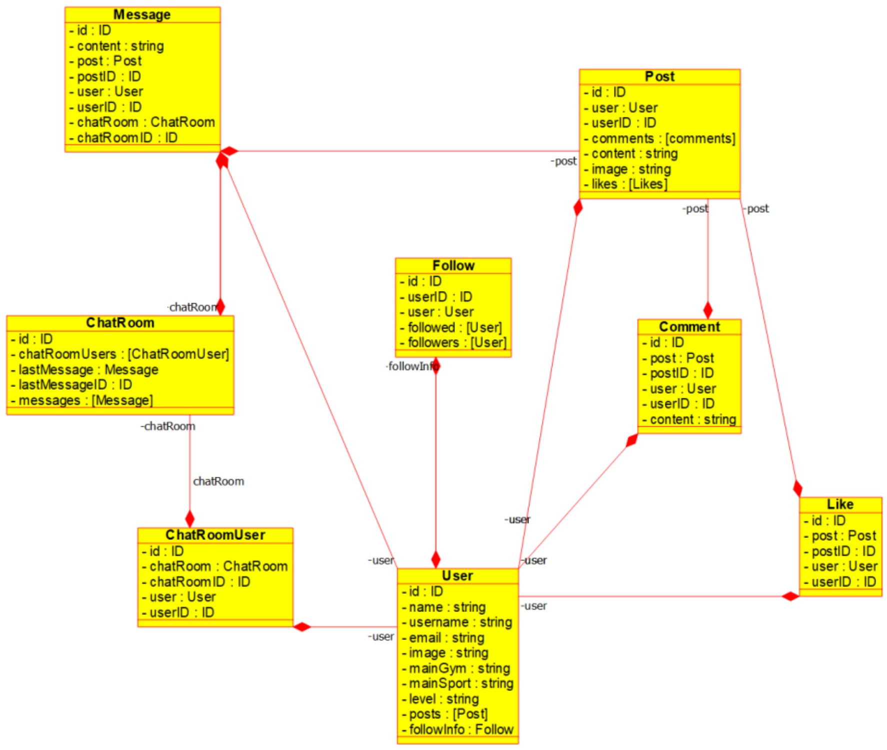


Figure 26: UML diagram of Ath-Meet GraphQL schema as generated by GraphQL Voyager schema visualizer [8].

Every model in the schema is attached to the User model that owns it by storing the User’s userID. The userID is a user’s primary key, and can be used by GraphQL to query the User’s data. The ChatRoom model does not link to the User directly because they have a many-to-many relationship, so instead it uses ChatRoomUsers to make the connection with Users, as shown in Figure 26. This allows us to query the ChatRoomUsers by User and by ChatRoom. GraphQL allows for a maximum of two index keys attached to a model for queries, so querying models with many-to-many relationships will return data sets with high cardinality [9]. Also, this method is used for group chats, and improves scalability of the application since the User model does not need to have an additional key for ChatRoom queries.

To connect with the DynamoDB database we have to add the GraphQL API to our project using Amplify and configure the API settings. Amplify gives the ability to start with a sample schema which can be modified to fit the project's requirements. The schema must be pushed to the server so Amplify will upload the schema to the cloud and create the appropriate DynamoDB tables for the schema. This will also generate the mutation, queries, and subscription code in the "src/graphql" folder. Once the files are imported they can be used freely in the source code to modify the database when using the application.

Using GraphQL is straightforward since the functions have names that define their abilities. There main types of queries and mutations that are used in GraphQL are create, get, list, update, and delete, and are used on the models as defined by the schema.

3.3.4 DynamoDB Database and S3 Storage

DynamoDB is a key-value and document NoSQL database provided by AWS . It stores all Ath-Meet content and holds tables that store the users, posts, likes, comments, and chat rooms. Each record in the database holds a unique identifier that is used as a primary key to reference the record with. The records can have other keys to connect with other tables on the database as is defined by the GraphQL schema. With each post or update in the application, a query or mutation is made through the GraphQL API. Only users authorized by the Cognito service can create and view data. Users are not allowed to delete data, since this is a permission granted only to administrator users in the AWS IAM.

The list of tables in the database are:

- ChatRoom: stores ChatRoomUsers and Messages
- ChatRoomUser: stores Users and ChatRooms
- Comment: stores Posts, Users, and comment content

- Follow: stores Users
- Like: stores likes in Post
- Message: stores messages in ChatRoom
- Post: stores user posts
- User: stores user profiles

Each table except for the ChatRoom table is linked to the User table through the User primary key. The ChatRoom table links to the User table through the ChatRoomUser table, and links the message table as well. AWS provides cloud storage for files through AWS Simple Storage Solution (S3). Ath-Meet uses S3 to store multimedia data such as the images users upload for posts, or profile pictures. S3 holds items in storage buckets whose access is managed through Cognito's Identity pools, and only authorized users of the application are allowed to upload and read data. To create the storage bucket and add access to storage in the project we use the Amplify command "amplify add storage" which will ask for the bucket name, region, who should have access and if they will be able to read, upload or delete data. Next, the changes must be pushed to the cloud using "amplify push" so that the services can be set up in the AWS system and managed through the AWS console. Pushes must be done after each schema change in order to update the resources on the servers. AWS restrict bucket access by default, so in order for the users to access the bucket, the Cognito Userpool must be given access privileges to the S3 bucket. This is done by creating a Cognito Federated Identity pool with policies that grant access to the bucket, and registering the Userpool under it.

3.4 Classes and Functions

3.4.1 Home Tab

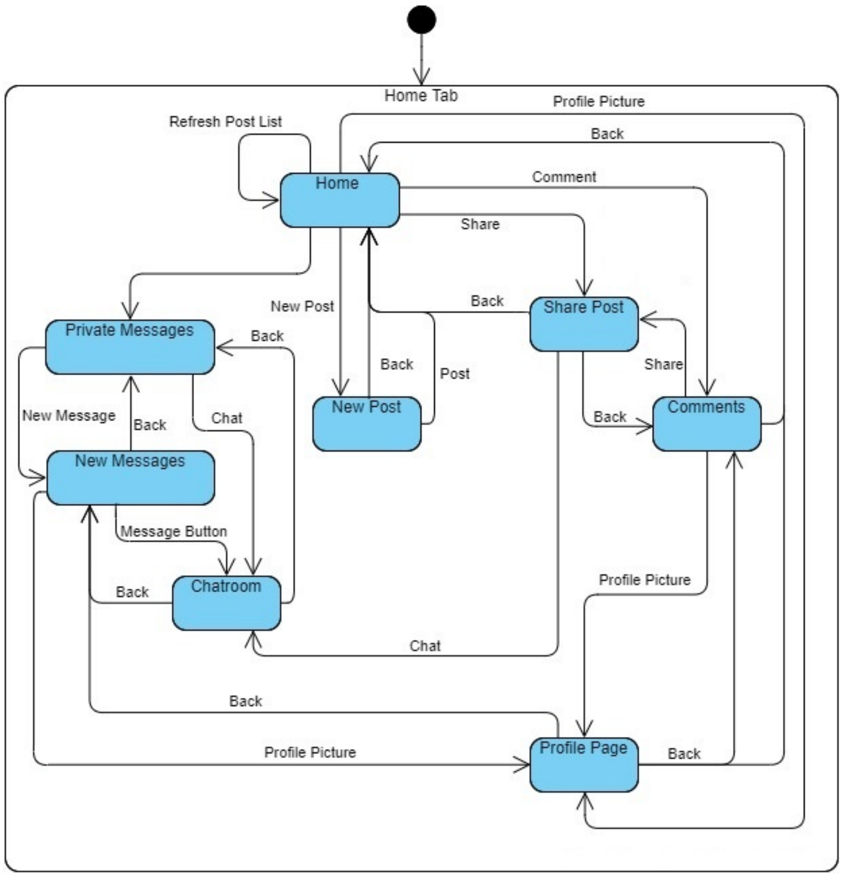


Figure 27: Home Tab State Diagram

The Home tab is the tab that users see first. It connects to the private messages, new post, share post, and comments screens directly. From the private messages screen, users can navigate to their chatrooms with other users, or create new ones in the new messages screen, as seen in Figure 27.

3.4.2 Creating Posts

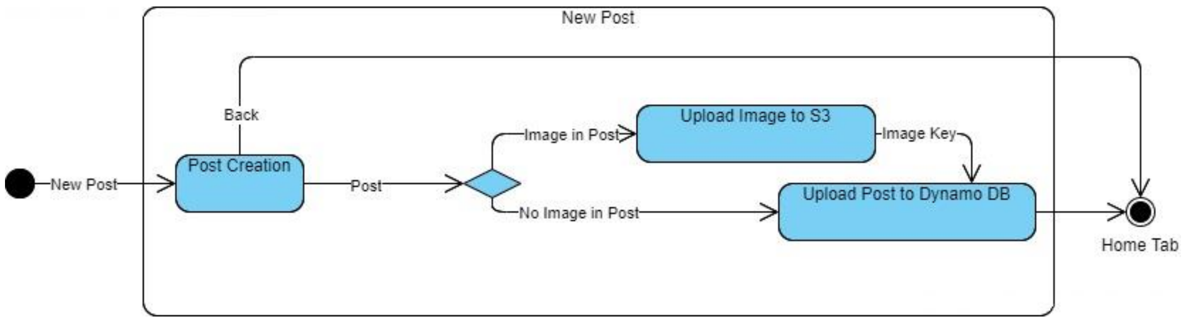
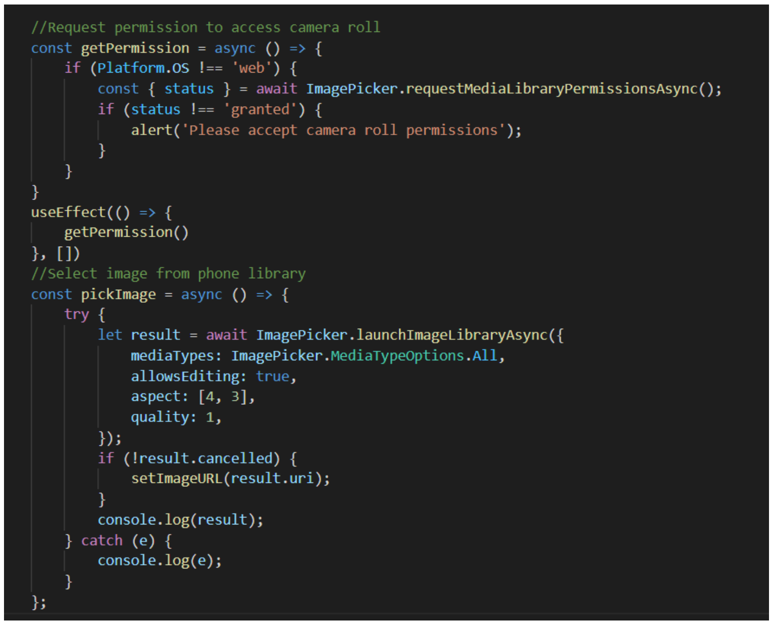


Figure 28: New Post Action Diagram

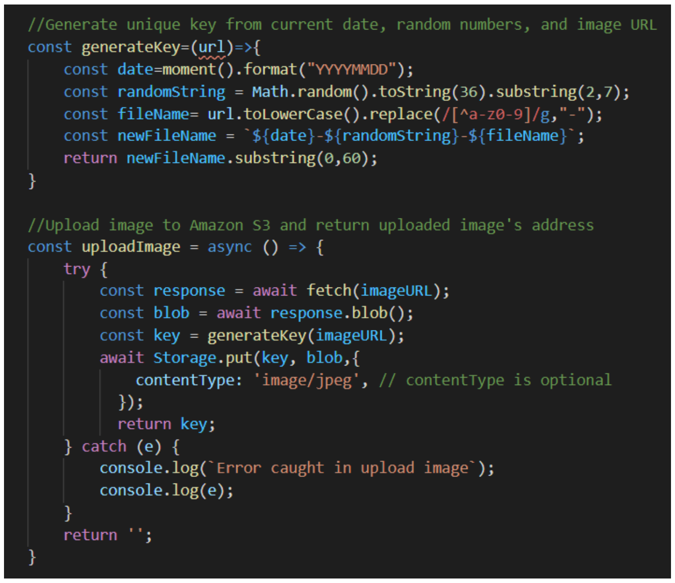
To create a post, we must save the content of the post to DynamoDB, and be able to select and upload an image to S3, as seen in the action diagram in Figure 28. Uploading images to S3 requires the use of an imported Image picker package from Expo, and for this project we chose to install the expo-image-picker package.



```
//Request permission to access camera roll
const getPermission = async () => {
    if (Platform.OS !== 'web') {
        const { status } = await ImagePicker.requestMediaLibraryPermissionsAsync();
        if (status !== 'granted') {
            alert('Please accept camera roll permissions');
        }
    }
}
useEffect(() => {
    getPermission()
}, [])
//Select image from phone library
const pickImage = async () => {
    try {
        let result = await ImagePicker.launchImageLibraryAsync({
            mediaTypes: ImagePicker.MediaTypeOptions.All,
            allowsEditing: true,
            aspect: [4, 3],
            quality: 1,
        });
        if (!result.cancelled) {
            setImageURL(result.uri);
        }
        console.log(result);
    } catch (e) {
        console.log(e);
    }
};
```

Figure 29: getPermission() and pickImage() in screens/NewPostScreen.tsx

To use the image picker, the application has to ask for permission through ImagePicker.requestMediaLibraryPermissionsAsync() in getPermission(), which allows the use of ImagePicker.launchImageLibraryAsync(). The function sets the selected images file path to the imageURL variable which is used to retrieve the image for when the post is being uploaded, as shown in Figure 29.

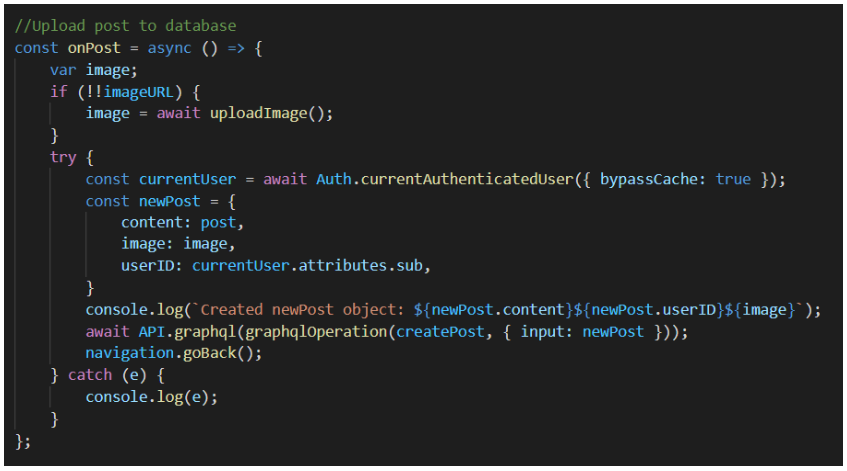


```
//Generate unique key from current date, random numbers, and image URL
const generateKey=(url)=>{
    const date=moment().format("YYYYMMDD");
    const randomString = Math.random().toString(36).substring(2,7);
    const fileName= url.toLowerCase().replace(/[^a-z0-9]/g,"-");
    const newFileName = `${date}-${randomString}-${fileName}`;
    return newFileName.substring(0,60);
}

//Upload image to Amazon S3 and return uploaded image's address
const uploadImage = async () => {
    try {
        const response = await fetch(imageURL);
        const blob = await response.blob();
        const key = generateKey(imageURL);
        await Storage.put(key, blob,{
            contentType: 'image/jpeg', // contentType is optional
          });
          return key;
    } catch (e) {
        console.log(`Error caught in upload image`);
        console.log(e);
    }
    return '';
}
```

Figure 30: uploadImage() and generateKey() in screens/NewPostScreen.tsx

To store objects in S3, a storage key must be provided, and the object must be converted into a Binary Large Object (blob) format [10]. Blobs are collections of binary data that are used in databases to store multimedia objects. The generateKey() function generates a unique key using the current date, random numbers, and the file path. “Storage” is part of the aws-amplify package, and Storage.put() takes the image’s key and blob as parameters and uploads the image to S3 using the AWS credentials that the user has received from Cognito. If the upload is successful, uploadImage() will return the image’s key, as shown in Figure 30.

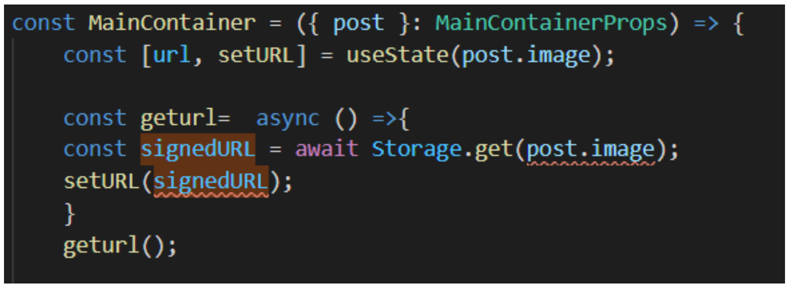


```
//Upload post to database
const onPost = async () => {
    var image;
    if (!!imageURL) {
        image = await uploadImage();
    }
    try {
        const currentUser = await Auth.currentAuthenticatedUser({ bypassCache: true });
        const newPost = {
            content: post,
            image: image,
            userID: currentUser.attributes.sub,
        }
        console.log(`Created newPost object: ${newPost.content}${newPost.userID}${image}`);
        await API.graphql(graphqlOperation(createPost, { input: newPost }));
        navigation.goBack();
    } catch (e) {
        console.log(e);
    }
};
```

Figure 31: Creating new post in onPost() in screens/NewPostScreen.tsx

When the user presses the “post” button, the function onPost() will be called to upload the image to S3 and to send the post’s details to DynamoDB. If there is an image, it will be uploaded first, and its key will be stored. The post’s details are stored in the newPost object, and will be uploaded to Dynamo by calling the createPost GraphQL operation with the newPost parameters which contains post data with the user Id and the image key, as shown in Figure 31.

3.4.3 Getting Images from S3



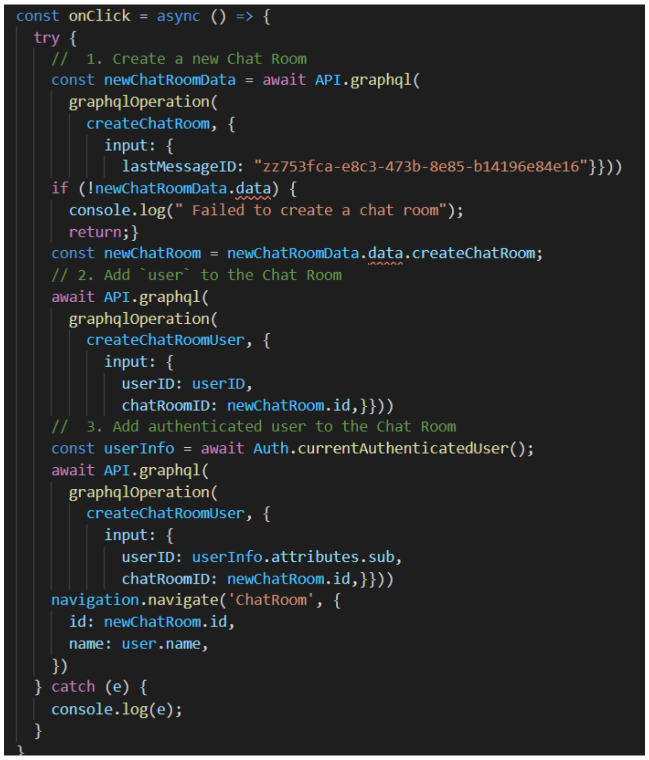
```
const MainContainer = ({ post }: MainContainerProps) => {
    const [url, setURL] = useState(post.image);

    const geturl=  async () =>{
    const signedURL = await Storage.get(post.image);
    setURL(signedURL);
    }
    geturl();
```

Figure 32: Getting signed URL of image from S3 in Post/MainContainer/index.tsx

Since AWS doesn’t grant public access to our bucket, the application must get a signed URL that can be used to access the objects. For it, we must use the Storage.get() function from aws-amplify, and pass the object’s key as the parameters. This will return a signed URL that can be used as a normal image URL to access the objects in our S3 bucket.

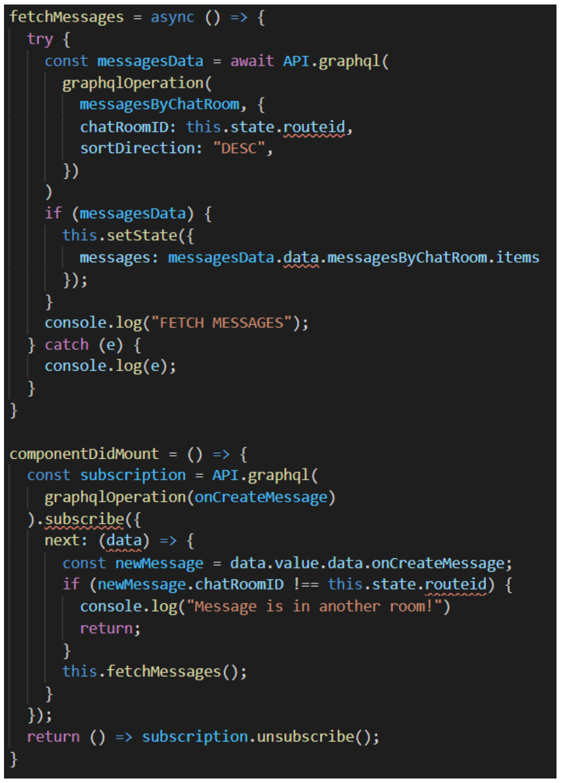
3.4.4 Chatrooms



```
const onClick = async () => {
  try {
    //  1. Create a new Chat Room
    const newChatRoomData = await API.graphql(
      graphqlOperation(
        createChatRoom, {
          input: {
            lastMessageID: "zz753fca-e8c3-473b-8e85-b14196e84e16"}}))
    if (!newChatRoomData.data) {
      console.log(" Failed to create a chat room");
      return;}
    const newChatRoom = newChatRoomData.data.createChatRoom;
    // 2. Add `user` to the Chat Room
    await API.graphql(
      graphqlOperation(
        createChatRoomUser, {
          input: {
            userID: userID,
            chatRoomID: newChatRoom.id,}}))
    //  3. Add authenticated user to the Chat Room
    const userInfo = await Auth.currentAuthenticatedUser();
    await API.graphql(
      graphqlOperation(
        createChatRoomUser, {
          input: {
            userID: userInfo.attributes.sub,
            chatRoomID: newChatRoom.id,}}))
    navigation.navigate('ChatRoom', {
      id: newChatRoom.id,
      name: user.name,
    })
  } catch (e) {
    console.log(e);
  }
}
```

Figure 33: onClick() in components/GoToChatButton/index.tsx

The GoToChatButton located in the "components" folder creates new chats when pressed. When pressed, it calls onClick() which calls the createChatRoom GraphQL operation that creates a new ChatRoom object in DynamoDB, and it sets the last message to a predefined message ID. The newly created ChatRoom's data is returned by the operation and stored in the newChatRoomData constant. Two ChatRoomUser objects for the user and the person they are messaging are subsequently created in the database by calling the createChatRoomUser GraphQL operation and passing each user's userID along with the ID of the newly created ChatRoom. Once the ChatRoom and ChatRoomUsers have been created, the function navigates to the ChatRoomScreen and passes the new ChatRoom's ID to the screen through the navigation route, as shown in Figure 33.

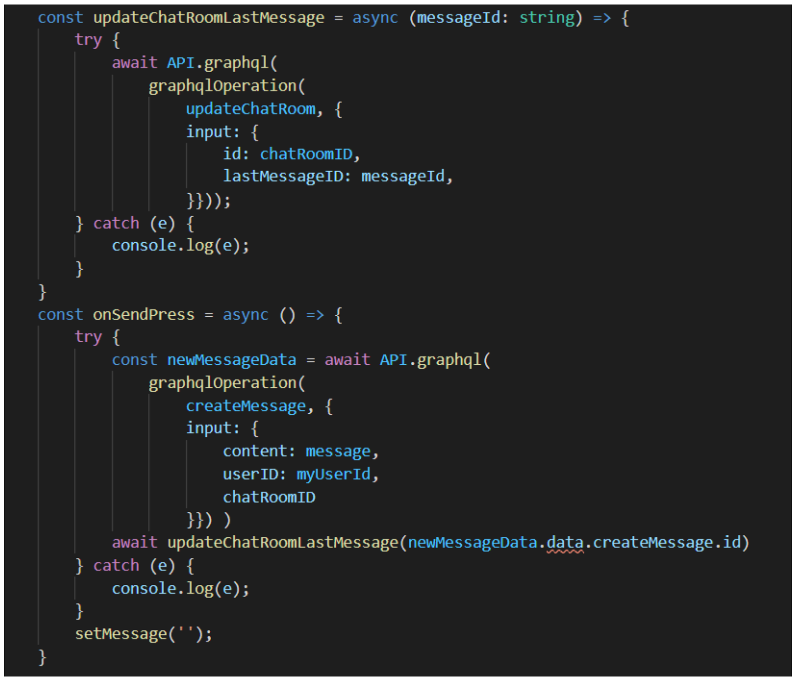


```
fetchMessages = async () => {
  try {
    const messagesData = await API.graphql(
      graphqlOperation(
        messagesByChatRoom, {
        chatRoomID: this.state.routeid,
        sortDirection: "DESC",
      })
    )
    if (messagesData) {
      this.setState({
        messages: messagesData.data.messagesByChatRoom.items
      });
    }
    console.log("FETCH MESSAGES");
  } catch (e) {
    console.log(e);
  }
}

componentDidMount = () => {
  const subscription = API.graphql(
    graphqlOperation(onCreateMessage)
  ).subscribe({
    next: (data) => {
      const newMessage = data.value.data.onCreateMessage;
      if (newMessage.chatRoomID !== this.state.routeid) {
        console.log("Message is in another room!")
        return;
      }
      this.fetchMessages();
    }
  });
  return () => subscription.unsubscribe();
}
```

Figure 34: fetchMessages() and GraphQL subscription in screens/ChatRoomScreen.tsx

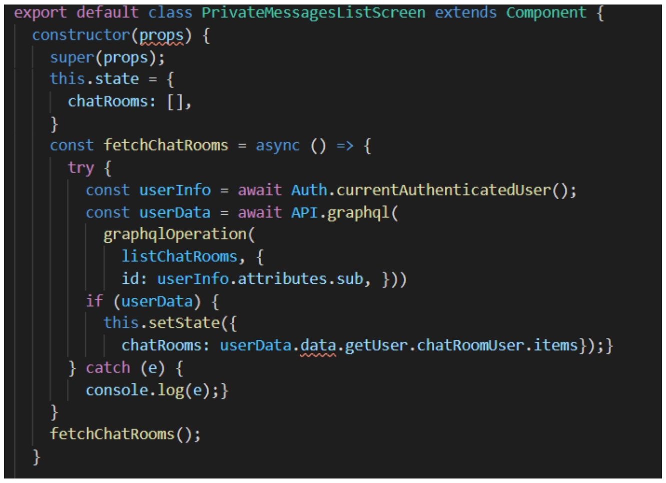
The ChatRoomScreen.tsx holds the private messaging chatrooms. As seen in Figure 34, messages are retrieved in fetchMessages() by using the messagesByChatRoom GraphQL operation, which lists the messages connected to the chatroomID that was received from the previous screen. This screen uses a GraphQL subscription to the chatroom in DynamoDB to update the message list as soon as new data is created in the chatroom. When new data is created, the subscription calls fetchMessages() which refreshes the screen to show the new messages.



```
const updateChatRoomLastMessage = async (messageId: string) => {
    try {
        await API.graphql(
            graphqlOperation(
                updateChatRoom, {
                input: {
                    id: chatRoomID,
                    lastMessageID: messageId,
                }}));
    } catch (e) {
        console.log(e);
    }
}
const onSendPress = async () => {
    try {
        const newMessageData = await API.graphql(
            graphqlOperation(
                createMessage, {
                input: {
                    content: message,
                    userID: myUserId,
                    chatRoomID
                }}) )
        await updateChatRoomLastMessage(newMessageData.data.createMessage.id)
    } catch (e) {
        console.log(e);
    }
    setMessage('');
}
```

Figure 35: components/InputBox/index.tsx

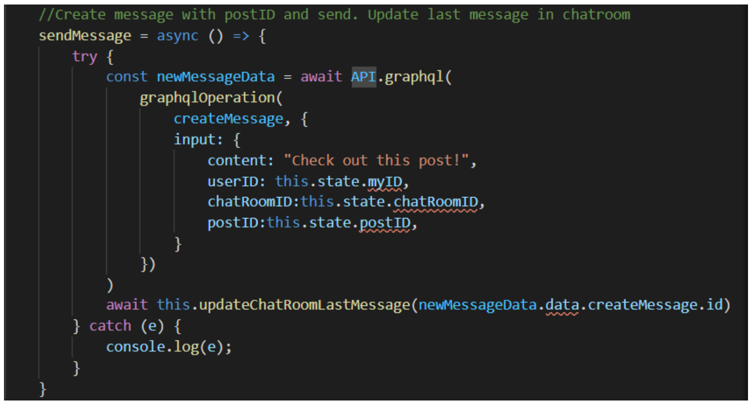
To send messages to the chatroom, users type into the InputBox component which stores their message and uploads it when the "send" button is pressed. Pressing the button calls the function onSendPress() which uses the createMessage GraphQL operation to create a new message in the chatroom table in DynamoDB, and updateChatRoomLastMessage() to set the last message of the chatroom as the one that was just created, as shown in Figure 35.



```
export default class PrivateMessagesListScreen extends Component {
  constructor(props) {
    super(props);
    this.state = {
      chatRooms: [],
    }
    const fetchChatRooms = async () => {
      try {
        const userInfo = await Auth.currentAuthenticatedUser();
        const userData = await API.graphql(
          graphqlOperation(
            listChatRooms, {
            id: userInfo.attributes.sub, }))
        if (userData) {
          this.setState({
            chatRooms: userData.data.getUser.chatRoomUser.items});}
      } catch (e) {
        console.log(e);}
    }
    fetchChatRooms();
  }
```

Figure 36: Fetching chatrooms in screens/PrivateMessagesListScreen.tsx

Once a ChatRoom object is created in the database, it will be visible in the users' private messages. The private messages are stored in the PrivateMessagesListScreen.tsx which retrieves all the ChatRoom objects connected to the user's ID, as seen in Figure 36.



```
//Create message with postID and send. Update last message in chatroom
sendMessage = async () => {
    try {
        const newMessageData = await API.graphql(
            graphqlOperation(
                createMessage, {
                input: {
                    content: "Check out this post!",
                    userID: this.state.myID,
                    chatRoomID:this.state.chatRoomID,
                    postID:this.state.postID,
                }
            })
        )
        await this.updateChatRoomLastMessage(newMessageData.data.createMessage.id)
    } catch (e) {
        console.log(e);
    }
}
```

Figure 37: Sending a post in sendMessage() in components/SendPostButton/index.tsx

Since our schema defines a message with an optional postID, posts can also be sent as messages through the ShareButton component. The "share" button is placed at the footer of each post, and when pressed will navigate to PostSendScreen.tsx. This screen shows a list of the user's existing chatrooms using the same method as the one for private messages, but in this screen there is a "send post" button attached to each chatroom in the list. When pressed, this button will send a post to the chatroom by calling sendMessage() in components/SendPostButton/index.tsx, as shown in Figure 37. This function creates a new message in the same way as it was described above with the ID of the post. The new message is created with a predefined message as its content and stored in the database. The last message in the chatroom is also updated to be the newly created message, and the message can be displayed with the post inside the chatroom screen.

3.4.5 Profile Tab

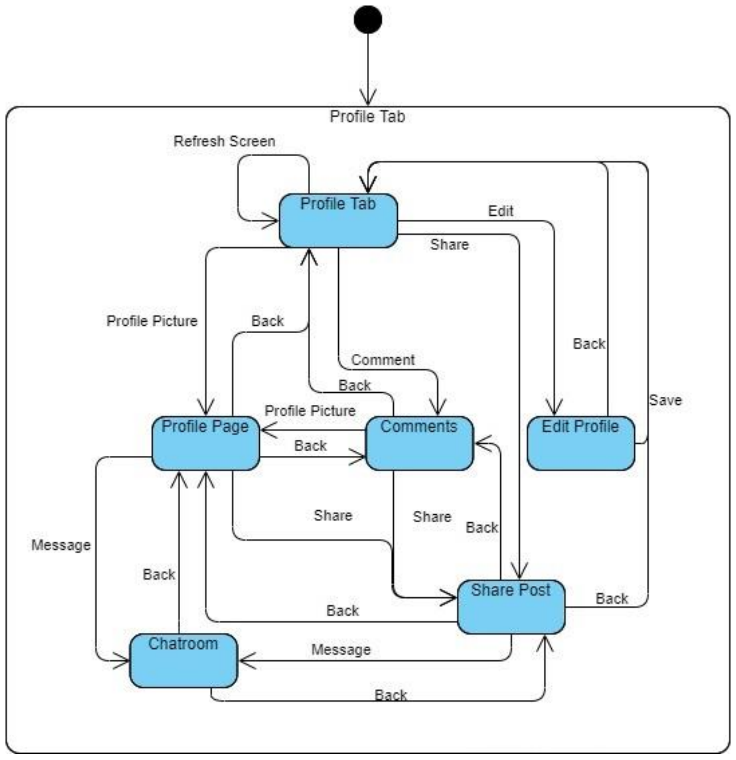


Figure 38: Profile Tab State Diagram

The Profile tab is where the user can go to access their personal profile page, and make changes to it. This includes uploading new profile pictures to S3, and updating the user's record in the User table in DynamoDB.

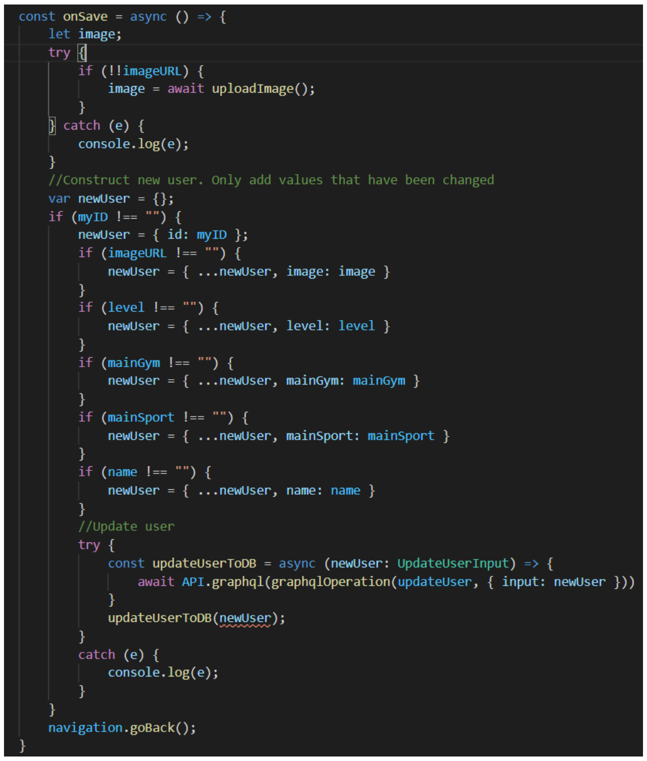

```
const onSave = async () => {
    let image;
    try {
        if (!!imageURL) {
            image = await uploadImage();
        }
    } catch (e) {
        console.log(e);
    }
    //Construct new user. Only add values that have been changed
    var newUser = {};
    if (myID !== "") {
        newUser = { id: myID };
        if (imageURL !== "") {
            newUser = { ...newUser, image: image }
        }
        if (level !== "") {
            newUser = { ...newUser, level: level }
        }
        if (mainGym !== "") {
            newUser = { ...newUser, mainGym: mainGym }
        }
        if (mainSport !== "") {
            newUser = { ...newUser, mainSport: mainSport }
        }
        if (name !== "") {
            newUser = { ...newUser, name: name }
        }
        //Update user
        try {
            const updateUserToDB = async (newUser: UpdateUserInput) => {
                await API.graphql(graphqlOperation(updateUser, { input: newUser }))
            }
            updateUserToDB(newUser);
        }
        catch (e) {
            console.log(e);
        }
    }
    navigation.goBack();
}
```

Figure 39: Constructing User object for updates in onSave() in screens/EditProfileScreen.tsx

In the edit profile screen, users have the option to change their profile picture, profile name, username, main gym, main sport, and experience level. The profile picture is selected and uploaded using the Expo image picker in the same way it is done in the “new post” button. The levels are predefined and can be selected from a dropdown menu created by using the RNPickerSelect Expo package. The level options are “Beginner,” “Intermediate,” “Advanced,” and “Expert.” Each attribute that the user can edit is stored in its own variable which is set to an empty string by default, and when the user presses the “save” button, onSave() is called. onSave() checks if the user has selected a new profile picture and tries to upload it. After the image is uploaded, a newUser object is created and initialized with the user’s userID. Several conditional statements check each profile attribute for changes, and attributes that have been changed are stored in the newUser object and then the updateUser GraphQL operation is called with the newUser object as its parameters, as shown in Figure 39.

3.4.6 Search Tab

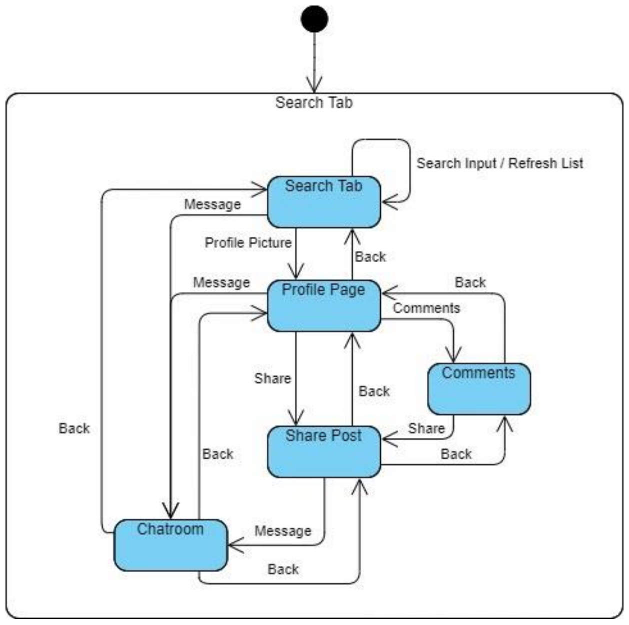
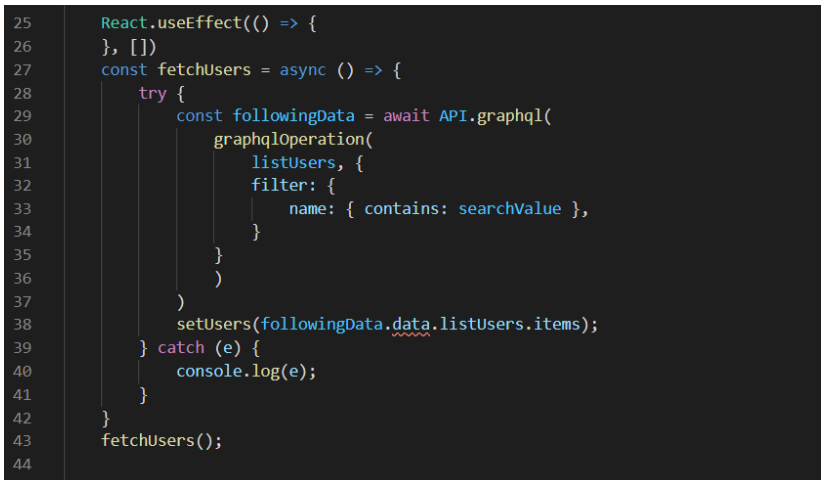


Figure 40: Search Tab State Diagram



```
React.useEffect(() => {
}, [])
const fetchUsers = async () => {
    try {
        const followingData = await API.graphql(
            graphqlOperation(
                listUsers, {
                filter: {
                    name: { contains: searchValue },
                }
            }
            )
        )
        setUsers(followingData.data.listUsers.items);
    } catch (e) {
        console.log(e);
    }
}
fetchUsers();
```

Figure 41: listUsers() GraphQL operation in screens/SearchScreen.tsx

The Search tab is the simplest tab in the application. It is only used to show the user a list of profiles that contain the name they are typing in the search bar. Under the search bar is a list of the matching user profiles which can be used to navigate to the user’s profile page. The list also lets the user go directly to a chatroom to send a message to a profile, as seen in Figure 40. The Search tab uses a filter on the listUser GraphQL operation to fetch users names that match what the user has typed into the search bar, as shown in Figure 41.

3.4.7 AthleteFinder Tab

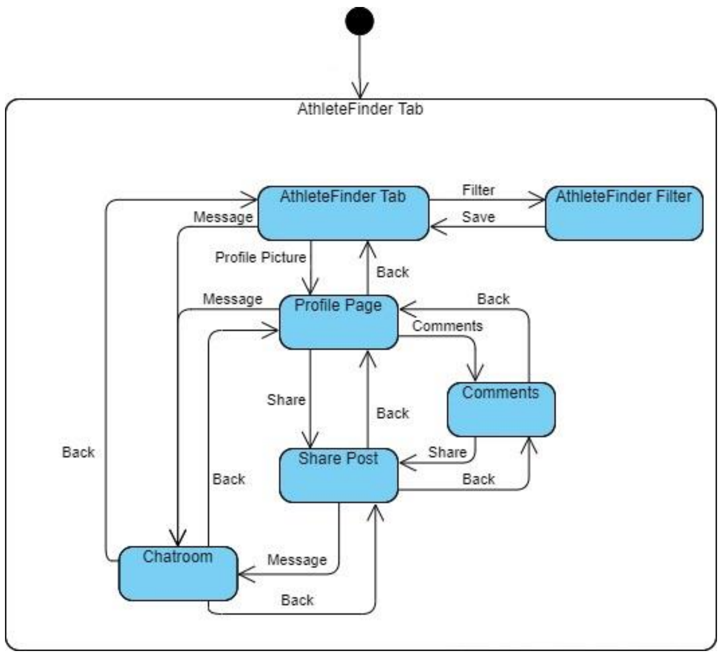
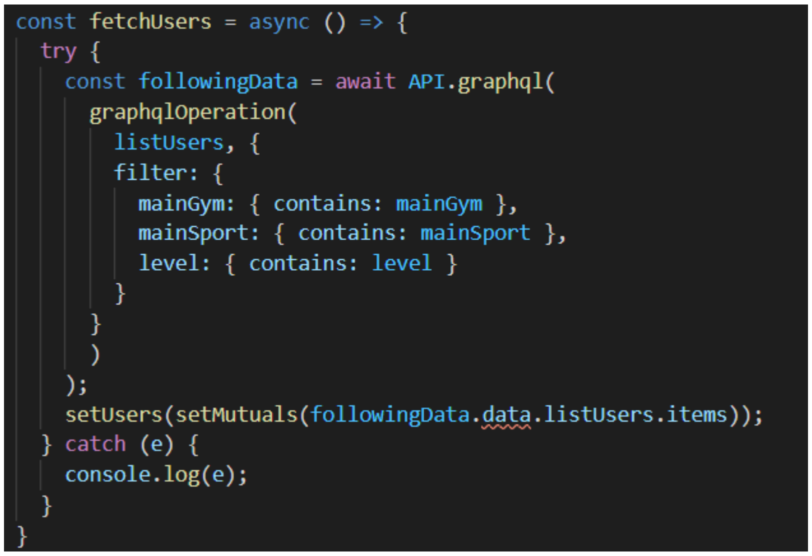


Figure 42: AthleteFinder Tab State Diagram

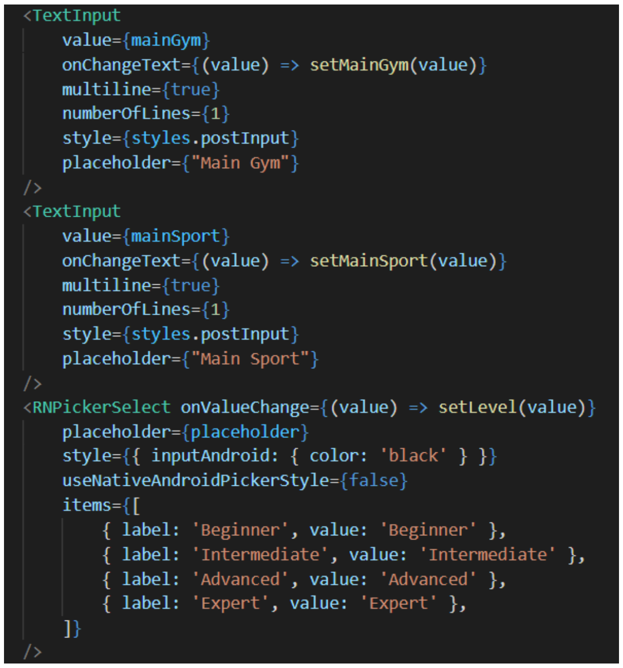
The AthleteFinder tab follows a similar layout to the search tab, and has the “filter” button that leads to the AthleteFinder filter screen, as seen in Figure 42. The AthleteFinder retrieves users by following a method similar to the one used by the Search tab.



```
const fetchUsers = async () => {
  try {
    const followingData = await API.graphql(
      graphqlOperation(
        listUsers, {
        filter: {
          mainGym: { contains: mainGym },
          mainSport: { contains: mainSport },
          level: { contains: level }
        }
      }
      )
    );
    setUsers(setMutuals(followingData.data.listUsers.items));
  } catch (e) {
    console.log(e);
  }
}
```

Figure 43: fetchUsers() in AthleteFinderScreen.tsx

Figure 43 shows the fetchUsers function in AthleteFinderScreen.tsx which fetches a list of users through the listUsers GraphQL operation with filters on the main gym, sport, and level attributes.

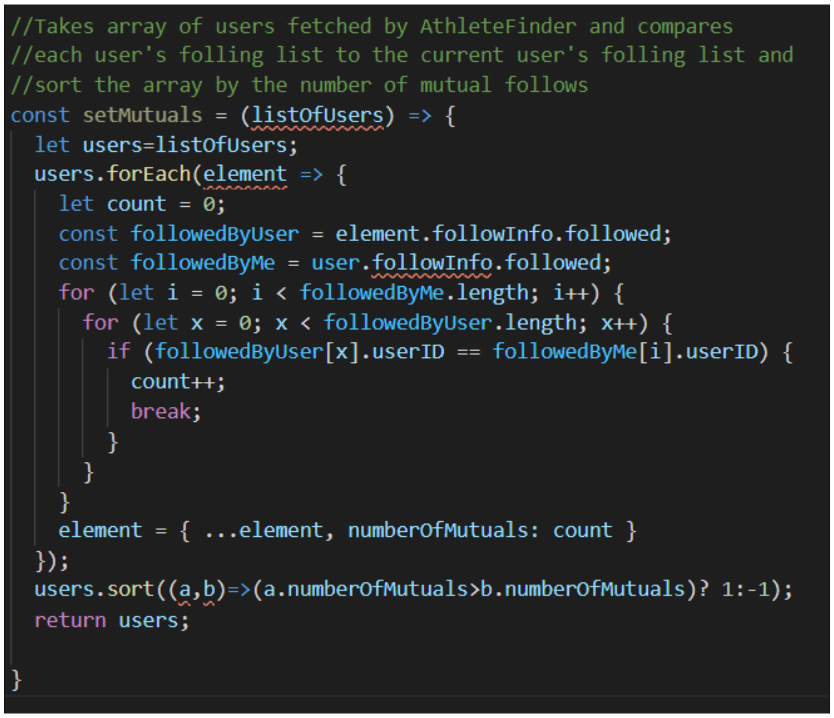


```
<TextInput
    value={mainGym}
    onChangeText={(value) => setMainGym(value)}
    multiline={true}
    numberOfLines={1}
    style={styles.postInput}
    placeholder={"Main Gym"}
/>
<TextInput
    value={mainSport}
    onChangeText={(value) => setMainSport(value)}
    multiline={true}
    numberOfLines={1}
    style={styles.postInput}
    placeholder={"Main Sport"}
/>
<RNPickerSelect onValueChange={(value) => setLevel(value)}
    placeholder={placeholder}
    style={{ inputAndroid: { color: 'black' } }}
    useNativeAndroidPickerStyle={false}
    items={[
        { label: 'Beginner', value: 'Beginner' },
        { label: 'Intermediate', value: 'Intermediate' },
        { label: 'Advanced', value: 'Advanced' },
        { label: 'Expert', value: 'Expert' },
    ]}
/>
```

Figure 44:AthleteFinder Filter Inputs and RNPickerSelect in AthleteFinderFilterScreen.tsx

These filter settings can be changed in the filter screen which takes in text inputs for the gym and sport, and allows the user to choose an experience level from a dropdown list.

To create this dropdown list, the “RNPickerSelect” module had to be installed and imported to the project, and its implementation can be seen in Figure 44. The level options are preset so that users can choose one of four options, or choose the placeholder which leaves the level value blank. Attributes that have been left blank will be set to a null value and do not become a factor in the search criteria. Once the “save” button is pressed, the filter settings are passed back to the AthleteFinder screen through the navigation route which refreshes the listed users according to the new filter criteria.



```
//Takes array of users fetched by AthleteFinder and compares
//each user's folling list to the current user's folling list and
//sort the array by the number of mutual follows
const setMutuals = (listOfUsers) => {
  let users=listOfUsers;
  users.forEach(element => {
    let count = 0;
    const followedByUser = element.followInfo.followed;
    const followedByMe = user.followInfo.followed;
    for (let i = 0; i < followedByMe.length; i++) {
      for (let x = 0; x < followedByUser.length; x++) {
        if (followedByUser[x].userID == followedByMe[i].userID) {
          count++;
          break;
        }
      }
    }
    element = { ...element, numberOfMutuals: count }
  });
  users.sort((a,b)=>(a.numberOfMutuals>b.numberOfMutuals)? 1:-1);
  return users;

}
```

Figure 45: setMutuals() in AthleteFinderScreen.tsx

The AthleteFinder prioritizes users that follow the same profiles that the user follows. In AthleteFinderScreen.tsx, the list of users found by the filter criteria are passed to setMutuals() which uses a forEach loop to go through the list, and compare each of the users to the current user. For both users, the list of people being followed is taken and the number of people found in both lists are counted and set in the numberOfMutuals variable. At the end of the function, the entire list is sorted in descending order of the numberOfMutuals, as seen in Figure 45. The people who have more in common with the user will show up higher on the list of results in their AthleteFinder searches.

3.5 Test Cases

3.5.1 Front End Test Cases

Due to the React Native being structured by nested components, manual front end testing and debugging can be effectively done through unit and integration testing. By setting checkpoints in the components and using console logs to output data and messages, the flow of the program can be easily monitored and assessed. Unit testing is effective because specific error messages are thrown in many processes by React Native which give valuable information on the steps needed to find and resolve issues. These error messages display a detailed list of the points where the errors occur which makes manual testing more efficient. When error messages are not helpful enough, logs showing the changes in data are used to find the moment of error which narrows down the possible problematic components. Integration testing after adding new components was also effective in finding errors before they escalate and cause major complications.

Bottom Tab Navigator		
Test	Goal	Outcome
Pressing Home tab	Navigates to Home tab screen	Successful
Pressing Search tab	Navigates to Search tab screen	Successful
Pressing AthleteFinder tab	Navigates to AthleteFinder tab screen	Successful
Pressing Profile tab	Navigates to Profile tab screen	Successful

Home Tab		
Test	Goal	Outcome
Pressing Messages Icon on the header	Navigates to private messages screen	Successful

Pressing New Post button	Navigates to new post screen	Successful
Pressing post’s Profile picture	Navigates to a profile page	Successful
Pressing Like button	Changes like button color	Successful
Pressing Share button	Navigates to screen showing list of chats	Successful
Pressing Comment button	Navigates to the comments screen	Successful

New Post Screen		
Test	Goal	Outcome
Pressing content text input box	Brings up keyboard	Successful
Pressing image text input box	Brings up keyboard	Successful
Pressing Post button	Navigates back to the Home tab. Clears new post screen	Successful
Pressing Back button	Navigates to Home tab	Successful

Private Messages Screen		
Test	Goal	Outcome
Pressing a Chat container	Navigates to Chatroom	Successful
Pressing New Message Button	Navigates to screen showing list of users	Successful
Pressing Back button	Navigates to Home tab	Successful

Chatroom Screen		
Test	Goal	Outcome
Pressing text input box	Brings up keyboard	Successful
Pressing Send Button	Clears text input box	Successful
Pressing Back button	Navigates to previous screen	Successful
Sending Post in message	Displays post in message container	Successful

Search Tab		
Test	Goal	Outcome
Pressing search bar	Brings up keyboard	Successful
Typing a username	Updates Users being displayed	Successful
Pressing Users Profile Picture	Navigates to profile page	Successful
Pressing Message button by the user	Navigates to Chatroom with the user	Successful

AthleteFinder Tab		
Test	Goal	Outcome
Pressing Filter button	Navigates to filter screen	Successful
Pressing Users Profile Picture	Navigates to targeted user's profile screen	Successful
Pressing Message button by the user	Navigates to Chatroom with the user	Successful

AthleteFinder Filter Screen		
Test	Goal	Outcome
Pressing Save button	Navigates to AthleteFinder tab and updates users	Failed. Navigation successful, but user list does not change
Pressing Users Profile Picture	Navigates to targeted user’s profile screen	Successful
Pressing Back button	Navigates to AthleteFinder tab	Successful

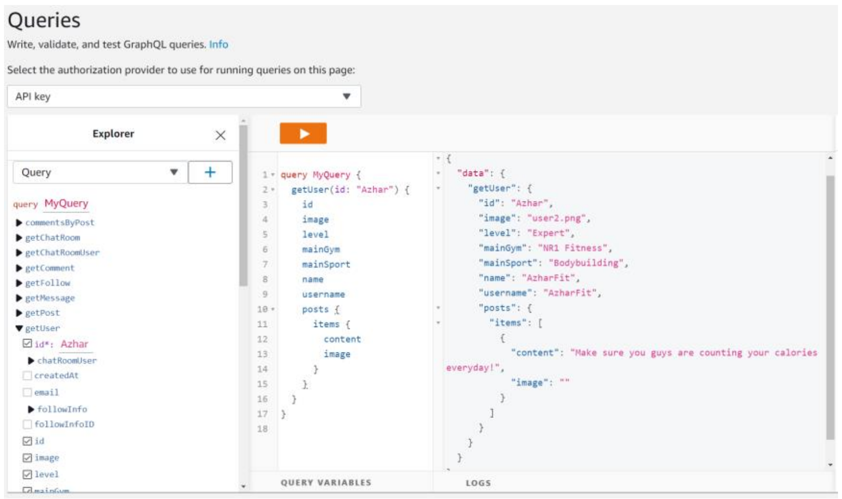
Profile Tab		
Test	Goal	Outcome
Pressing Edit Profile button	Navigates to edit profile screen	Successful
Pressing following	Navigates to screen showing list of users	Successful
Pressing followers	Navigates to screen showing list of users	Successful

Edit Profile Screen		
Test	Goal	Outcome
Pressing Save button	Navigates to Profile tab screen	Successful
Pressing Input boxes	Brings up keyboard	Successful
Pressing Level dropdown menu	Dropdown list appears	Successful
Pressing Back button	Navigates to Profile tab	Successful

User Profile Page		
Test	Goal	Outcome
Pressing Back button	Navigates to previous screen	Successful
Pressing Following button	Navigates to screen showing list of users	Successful
Pressing Followers button	Navigates to screen showing list of users	Successful
Pressing Follow button	Toggles between follow icon with check and follow icon with plus	Successful

3.5.2 Back End Test Cases

Back end tests can also be done through unit and integration testing, and AWS provides services that allow us to closely inspect and analyze backend components and services. The tests used in front end tests can be coupled with data logs to present the information received from our backend services for review. We can observe the changes in the information to detect the possible faults.



```
query MyQuery {
  getUser(id: "Azhar") {
    id
    image
    level
    mainGym
    mainSport
    name
    username
    posts {
      items {
        content
        image
      }
    }
  }
}
```

```
{
  "data": {
    "getUser": {
      "id": "Azhar",
      "image": "user2.png",
      "level": "Expert",
      "mainGym": "NR1 Fitness",
      "mainSport": "Bodybuilding",
      "name": "AzharFit",
      "username": "AzharFit",
      "posts": {
        "items": [
          {
            "content": "Make sure you guys are counting your calories
everyday!",
            "image": ""
          }
        ]
      }
    }
  }
}
```

Figure 46: GraphQL Operations in the AWS Console for AppSync API

When the data does not match expectations, it is useful to observe the processes in the AWS console which gives us access to the data directly in the backend services. For example, in Figure 46 we can see the AWS AppSync API console which allows us to test our GraphQL operations on the servers. This can be helpful when operations in the application are not working as intended because it allows us to perform the operation in an environment independent of the application, and it gives us additional error messages that can be used to find and solve the problems. DynamoDB and S3 also have their own consoles that allow for direct access to their data.

Authentication Screens		
Test	Goal	Outcome
Entering correct credentials to sign in	Logs user into the application	Successful
Entering incorrect credentials to sign in	Denies access to the application and displays message saying incorrect credentials	Successful
Entering appropriate details in sign up screen	Registers user in Cognito user pool	Successful
Entering username in forgot password	Sends email with confirmation code	Successful
Entering correct confirmation code and new password	Changes user's password	Successful

Home Tab		
Test	Goal	Outcome
Private Messages List Screen content	Displays list of existing chatrooms between user and other profiles	Successful
Pressing Profile pictures on posts	Navigates to Profile screen of the targeted user	Successful

Pressing Like button	Sends user’s like to the database to be saved under the targeted post	Successful
Pressing Share	Brings up tab showing messages to share to	Successful
Pressing Send Post button in share screen	Sends post as message to the chatroom	Successful
Pressing Comment button	Navigates to the comments screen with comments of targeted post	Successful

New Post Screen		
Test	Goal	Outcome
Pressing Post button	Posts the post to the database Posts table	Successful
Pressing Post button with attached image	Posts post, and saves image in AWS S3 database with its own URL	Successful

Chatroom Screen		
Test	Goal	Outcome
Pressing text input box	Brings up keyboard	Successful
Pressing Send Button	Clears text input box	Successful
Pressing Back button	Navigates to previous screen	Successful
Sending Post in message	Displays post in message container	Successful

Search Tab		
Test	Goal	Outcome
Typing name into the search bar	Retrieves list of users in User database table with name that contains the input	Successful

AthleteFinder Tab		
Test	Goal	Outcome
Calling AthleteFinder Function with Filter settings	Retrieves list of users in User database table with profile information that matches selected filter options	Successful

Profile Tab		
Test	Goal	Outcome
Pressing Sign out button	Signs user out of the application and requires Sign in	Successful

Edit Profile Screen		
Test	Goal	Outcome
Pressing Save after entering new information	Updates user’s profile in database with the newly given information	Successful
Selecting new profile picture and pressing save	Uploads image to S3 and updates user’s profile picture in the database	Successful

Other User’s Profile Page		
Test	Goal	Outcome
Pressing private message button	Creates new chat with the user	Successful
Pressing Follow button when user is not followed	Updates this user’s followInfo record by adding the target user to the “following” array. Adds this user to the target user’s “followers” array	Successful. User is added to the array.
Pressing Follow button when user is already followed	Updates this user’s followInfo record by removing the target user from the “following” array. Removes this user from the target user’s “followers” array	Failed. GraphQL operation fails to fetch accurate data.

3.6 Conclusions

The test cases provide a clear picture on the state of application and keep track of the project’s progress. These tests provide important data for the development of the application, since it narrows down the functions that must be examined. Specifically in the test cases for the AthleteFinder, we can conclude that the problem lies in sharing information across the React Native components since the GraphQL operations work when the filter’s settings are hard coded in. This means that the filter screen does not connect with the AthleteFinder tab properly, and that it is a frontend issue only. In this case, it will be important to investigate the function in AthleteFinderScreen.tsx that set’s the new filter options from AthleteFinderFilterScreen.tsx, and making it a feature that needs improvement. Another important case to examine is the retrieval of information about a user’s followed and following lists. We can see that user’s can be added into the arrays, however the queries return erroneous or corrupted data. Since the problem is not caused by the source code of the application or a problem with React Native, we can deduce that it is the result of a problem in our GraphQL schema. Upon closer inspection of the schema diagram in Figure 26, we can see that the Follow model in our schema is set up to store

User models in an array. This creates a many-to-many relationship that GraphQL cannot query properly, which results in the unusable data we received in our tests. Thus, the schema needs improvement and refactoring in such a way that the Follow model holds an indirect connection to the User model, and similar to the way the ChatRoom model connects to the User model.

Overall, the application functions as desired, and it achieves the task of creating a platform to share and interact with each other. The application lets users navigate to profiles easily, and the private messages allow user’s another avenue for communication. The sharing of posts privately or leaving public comments promotes socialization and discussion. Users are welcome to post about their own progress and achievements which will be available for all users to see on their profile pages. Additionally, people’s fitness information is constantly presented throughout the application which lets the user easily identify people that they may share an activity with, would like to train with, or would like to contact. All of these features help us achieve our goal of creating a social media application that promotes health and fitness, and encourages people to reach out to each other to train together.

4. Bibliography

[1] “Introduction · React Native.” *React Native*, reactnative.dev/docs/getting-started.

[2] “AWS Documentation.” *Amazon*, 20 04 2021, docs.aws.amazon.com/index.html?nc2=h_ql_doc_do

[3] “Introduction to Expo.” *Expo Documentation*, docs.expo.io/.

[4] “What Is Amazon Cognito?.” *Amazon*, 03 03 2021, docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html.

[5] Awati, Rahul, and David Carty. “What Is Amazon Cognito and How Does It Work?” *SearchAWS*, TechTarget, 1 05 2021, searchaws.techtarget.com/definition/Amazon-Cognito.

[6] “Amplify Libraries - Amplify Docs.” *Amplify Framework Documentation*, docs.amplify.aws/lib/q/platform/js.

[7] “GraphQL is the better REST.” *How to GraphQL*, www.howtographql.com/basics/1-graphql-is-the-better-rest/.

[8] “GraphQL Voyager.”*GraphQL Voyager*, apis.guru/graphql-voyager/.

[9] “GraphQL Relations.” *Fauna Documentation*, docs.fauna.com/fauna/current/api/graphql/relations.

[10] “Blob - Web APIs: MDN.” *Web APIs | MDN*, developer.mozilla.org/en-US/docs/Web/API/Blob.